BRICS

Basic Research in Computer Science

BRICS NS-99-3 Mosses & Watt (eds.): AS '99 Proceedings

Proceedings of the Second International Workshop on

Action Semantics AS '99

Amsterdam, The Netherlands, March 21, 1999

Peter D. Mosses
David A. Watt
(editors)

BRICS Notes Series NS-99-3

ISSN 0909-3206 May 1999

Copyright © 1999, BRICS, Department of Computer Science
University of Aarhus. All rights reserved.

Reproduction of all or part of this work
is permitted for educational or research use
on condition that this copyright notice is
included in any copy.

See back inner page for a list of recent BRICS Notes Series publications.
Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK–8000 Aarhus C
Denmark

Telephone:	+45 8942 3360
Telefax:	+45 8942 3255
Internet:	BRICS@brics.dk

BRICS publications are in general accessible through the World Wide Web and anonymous FTP through these URLs:

`http://www.brics.dk`
`ftp://ftp.brics.dk`
This document in subdirectory `NS/99/3/`

AS'99

Second International Workshop on Action Semantics

Proceedings

Amsterdam, The Netherlands

21st March 1999

Foreword

Action Semantics[1] is a practical framework for formal semantic description of programming languages. Since its appearance in 1992, action semantics has been used to describe major languages such as Pascal, SML, ANDF, and Java, and various tools for processing action semantic descriptions have been developed. Recently, the close relationship between action semantics and monadic approaches to denotational semantics has been established.

The workshop was held as a satellite event of ETAPS'99 in Amsterdam. It attracted 18 participants from a total of 10 different countries, several of whom came to Amsterdam especially to attend the workshop. (The Brazilian participants were unfortunately unable to attend, but they provided an automated presentation of their work, which was shown at the workshop.)

As can be seen from the workshop programme and the contributed papers, much interesting work was presented and discussed during the one day. Special thanks to the invited speakers, Philipp Kutter and Alfonso Pierantonio, for presenting their recent work towards providing support for action semantics in Montages; and to all the authors for keeping closely to a tight schedule, not only when giving their talks during the workshop, but also when preparing their papers for this Proceedings volume.

The final discussion session revealed plans for a revised version of the action notation used in action semantics—taking into account the experiences gained since the first full version appeared in 1992—as well as for further development of support tools for action-semantic descriptions.

The workshop was generally regarded as a useful and productive event by the participants, and this Proceedings volume should help to disseminate the reported work to those who did not attend. Partly in view of the planned revision of action notation, a third workshop is to be held relatively soon—the tentative plan is for it to take place in May 2000, in Recife, Brazil, as a 2-day satellite event of the annual Brazilian Symposium on Programming Languages. Details will be announced on the mailing list `action-semantics@brics.dk`, which is also to be used for reporting results, coordinating projects, and discussing features of action semantics and related frameworks.

The organizers of AS'99 gratefully acknowledge funding and sponsorship from:

BRICS (Centre for Basic Research in Computer Science, Denmark)[2]

Computing Science Department, University of Glasgow, Scotland

Peter D. Mosses
BRICS & Dept. of Computer Science
Univ. of Aarhus, Denmark

David A. Watt
Computing Science Dept.
Univ. of Glasgow, Scotland

[1]Web page: `http://www.brics.dk/Projects/AS/`

[2]Established by the Danish National Research Foundation, in collaboration with the Universities of Aarhus and Ålborg.

Programme

08:30	Registration
09:00	**Invited talk:**
	Philipp Kutter, **Alfonso Pierantonio** (TIK, ETH Zürich, Switzerland) *Generating an Action Notation Environment from Montages Descriptions*
10:00	**Tools:**
	Stephan Diehl (Universität des Saarlandes, Saarbrücken, Germany) *Bootstrapped Semantics-Directed Compiler Generation*
10:30	Coffee
11:00	**Tools, continued:**
	Hermano Perrelli de Moura, **Luis Carlos de Sousa Menezes** (Federal University of Pernambuco, Recife, Brazil) *The Abaco System - An Algebraic Based Action Compiler*
11:30	**Kyung-Goo Doh**, **Hyun-Goo Kang** (Hanyang University, Korea) *Online Partial Evaluation of Actions*
12:00	**Kent D. Lee** (University of Iowa, USA) *Tuple Sort Inference in Action Semantics*
12:30	Lunch
14:00	**Recent action-semantic descriptions:**
	David A. Watt (University of Glasgow, Scotland) *The Static and Dynamic Semantics of SML*
14:30	**Deryck Brown** (The Robert Gordon University, Aberdeen, Scotland), **David A. Watt** (University of Glasgow, Scotland) *JAS: a Java Action Semantics*
15:00	Coffee
15:30	**Theoretical foundations:**
	Peter D. Mosses (SRI International, Menlo Park, USA) *A Modular SOS for Action Notation*
16:00	**Søren B. Lassen** (University of Cambridge, England) *Towards a New Action Notation*
16:30	**Concluding discussion**
	The Future of Action Semantics
	(with position statements by Peter D. Mosses and David A. Watt)
17:30	Close

Contents

Generating an Action Notation Environment from Montages Descriptions

Matthias Anlauff[1], Philipp W. Kutter[2],
Alfonso Pierantonio[2], and Lothar Thiele[2]

[1] GMD FIRST, D-10000 Berlin
ma@first.gmd.de
[2] Federal Institute of Technology, CH-8092 Zürich
{kutter, alfonso, thiele}@tik.ee.ethz.ch

Abstract. In the present paper, a methodology is presented which enables the implementation of the Action Notation formalism based on a formal and modular specification. As a result, an interpreter and debugger is automatically generated which allows the visualization of an Action Notation program execution and the inspection of all semantic identities in terms of the given formal specification.
These results are based on several new concepts. At first, a formal description of Action Notation is provided by means of Montages. Montages are a semi-visual formalism for the specification of syntax and semantics of programming languages. Moreover, the structuring of Action Notation via facets is refined and used to define a new specification architecture that ensures the required modularity. The tool support for Montages (Gem-Mex) automatically generates a prototypical implementation from the language's Montages specification.

1 Introduction

Action Semantics [Mos92,Wat91] allows the description of large, realistic programming languages like Standard ML [Wat99] or Java [BW99]. The main reason is that the addition of new constructs to a described language does not require reformulation of the already-given description. According to [Mos98b,Mos98a] we call this property *modularity*. Nevertheless, experience showed that despite of modularity and other pragmatic qualities of Action Semantics, tool support is crucial when large languages are specified.

In Action Semantics, as in denotational semantics, semantic functions map the abstract syntax of the described language to semantic entities. Here, however, the semantic entities are so-called actions, which are specified in Action Notation (AN), rather than higher-order functions expressed in lambda-notation. In this paper, we describe how to obtain an environment for the execution and inspection of AN descriptions. Implementing such an environment by hand presents some disadvantages. For example, it is very difficult to maintain the consistency of such a system if the definition of AN evolves over time. Moreover, the implementation is not accessible for inspection to the user as there is no direct link (neither formally nor visually) between the specification of AN and the execution of a AN program.

Generating the environment from an executable specification solves the mentioned problems. For instance, the ASD tool [vDM96] is generated from an ASF+SDF description of AN. The resulting environment consists of a structural editor, a consistency checker, and semantics inspection support on the abstraction level of term rewriting. ASF+SDF, as well as Denotational Semantics, SOS, and Natural Semantics present the disadvantage of not being inherently modular [Mos98b,Mos98a].

If AN is extended, as for instance in [MM93], a modular semantics is necessary. Otherwise, for each extension of AN re-proving all the laws of AN is required. An approach to achieve modularity for this purpose has been presented in [Wan97].

Independent of modular proofs, it is useful to have a modular tool support that allows to quickly adopt the newest theoretical results and extend the existing implementations accordingly. In addition, it may be useful to test the practicability of an extension before doing all the work of re-proving the laws of AN. Another advantage of basing a tool on a modular description of AN is the possibility to develop, test, and validate the specification in small pieces. Once their correct behavior is assessed in isolation, one is able to investigate the interaction between modules, while putting everything together. Such a development process structures the resulting implementation and makes it more transparent and accessible for the user.

This work illustrates the first tool environment for AN based on and automatically generated from a modular specification. AN descriptions can be visualized, debugged, and interpreted in terms of the specification. This corresponds to *origin tracking*, i.e. to the direct and consistent relation of the implementation with the underlying specification.

The described results are based on the following techniques:

- We use a a semi-visual formalism for the specification of syntax and semantics of programming languages, called Montages [KP97a,AKP98]. Similar to Action Semantics, Montages aim at being a pragmatic framework for language engineering. Montages are based on context-free grammars (EBNF), finite state machines for visual control flow and Abstract State Machines (ASMs) [Gur88,Gur95] for the dynamic semantics. These concepts are informally described in Section 2.
- In Section 3, we introduce an architecture which is based on a refined partition of AN in facets. The imperative and parts of the basic facet are formalized, and as an example of the modularity it is shown how they can be combined. The given parts show and explain the techniques used in the complete Montages of Action Notation [AKP97c], covering the remaining parts of the basic facet as well as the declarative, and reflective facets. The communicative facet has not yet been incorporated.

– Finally, Section 4 gives an impression of the generated AN environment and shows how it serves as a platform for conducting empirical validation of the design decisions using origin tracking.

2 Montages

In this section, the methodology is introduced on which the results of the paper are based. Montages is a formalism for the specification of programming languages. We will describe informally only those features which are relevant for the specification of Action Notation. The complete specification of Montages is available in [KP97a,AKP98].

The aim of Montages is to document formally the decisions taken during the design process of realistic programming languages. Syntax, static and dynamic semantics are given in a uniform and coherent way by means of semi-visual descriptions. The static aspects of a language are diagrammatic descriptions of control flow graphs, and the overall specifications are similar in structure, length, and complexity to those found in common language manuals. The intended use of the tool Gem-Mex is, on one hand to allow the designer to 'debug' her/his semantics descriptions by empirical testing of whether the intended decisions have been properly formalized; on the other hand, to automatically generate a correct (prototype) implementation of programming languages from the description, including visualization and debugging facilities.

The departure point for our work has been the formal specification of the C language [GH93][1], which showed how the state-based formalism Abstract State Machines [Gur88,Gur95,Hug] (ASMs), formerly called Evolving Algebras, is well-suited for the formal description of the dynamic behavior of full-blown practical languages. In essence, ASMs constitute a formalism in which a state is updated in discrete time steps. Unlike most state-based systems, the state is given by an algebra, that is, a collection of functions and universes. The state transitions are given by rules that update functions pointwise and extend universes with new elements. The model presented in [GH93] describes the dynamic semantics of the C language by presuming on an explicit representation of *control and data flow as a graph* (CDG). This represents a major limitation for such a model, since the control and data flow graph is a crucial part of the specification. Therefore, we developed Montages which extend the approach in [GH93] by introducing a mapping which describes how to obtain the control and data flow graph starting from the abstract syntax tree.

The formulation of Montages [KP97a] was strongly influenced by some case studies where the Oberon language [KH95,KP97b] has been specified. Oberon is an object-oriented language that is used for the implementation of compilers, operating systems [WG92], various applications, and teaching [RW92]. Montages have been used also in other case studies, such as the specification of the

[1] Historically the C case-study was preceded and paralleled by work on Pascal [Gur88], Modula2, Prolog, and Occam, see [BH98] for a commented bibliography on ASM case studies.

Java [Wal97] language, the front-end for correct compiler construction [HLT98], and the design and prototyping of a domain-specific language in an industrial context [KST98]. The logical/algebraic characterization of an extended Montages formalism has been presented in [AKP98]. The tool Gem-Mex has been completely re-implemented with respect to the initial prototype [AKP97a]. Complete references, documentation and tools can be obtained via
http://www.tik.ee.ethz.ch/∼montages/.

The experience showed that the underlying model for the dynamic semantics, namely the specification of a control flow graph including conditional control flow and data flow arrows and its close relationship to the well known concept of *Finite State Machines*, shortens the learning curve considerably. It confers to the formalism enhanced pragmatic qualities, such as writability, extensibility, readability, and, in general, ease of maintenance.

In our formalism, the specification of a language consists of several components. As depicted in Fig. 1, the language specification is partitioned into three parts.

1. The EBNF production rules are used for the context-free syntax of the specified language L, and they allow to generate a parser for programs of L. Furthermore, the rules define in a canonical way the signature of abstract syntax trees (ASTs) and how the parsed programs are mapped into an AST. Section 2.1 contains the details of this mapping. In Fig. 1 the dotted arrow from the EBNF rules visualizes that this information is provided from the Montage language specification.
2. The next part of the specification is given using the *Montage Visual Language* (MVL). MVL has been explicitly devised to extend EBNF rules to finite state machines (FSM). A MVL description associated to an EBNF rule defines basically a *local* finite state machine and contains information how this FSM is plugged into the *global* FSM via an inductive decoration of the abstract syntax trees. To this end, each node is decorated with a copy of the finite state machine fragment given by its Montage. The reference to descendents in the AST defines an inductive construction of a global structured FSM. In Section 2.2 we define how this construction works exactly.
3. Finally, any node in the FSM may be associated with an Abstract State Machine (ASM) rule. This *action rule* is fired when the node becomes the current state of the FSM. As shown in Fig. 1, the specification of these rules is the third part of a Montages specification. The underlying abstract state machine formalism is shortly described in Section 2.3.

The complete language specification is structured in specification modules, called Montages. Each Montage is a "BNF-extension-to-semantics" in the sense that it specifies the context-free grammar rule (by means of EBNF), the (local) finite state machine (by means of MVL), and the dynamic semantics of the construct (by means of ASMs). The special form of EBNF rules allowed in a specification and the definition of Montages lead to the fact that each node in the abstract syntax tree belongs exactly to one Montage.

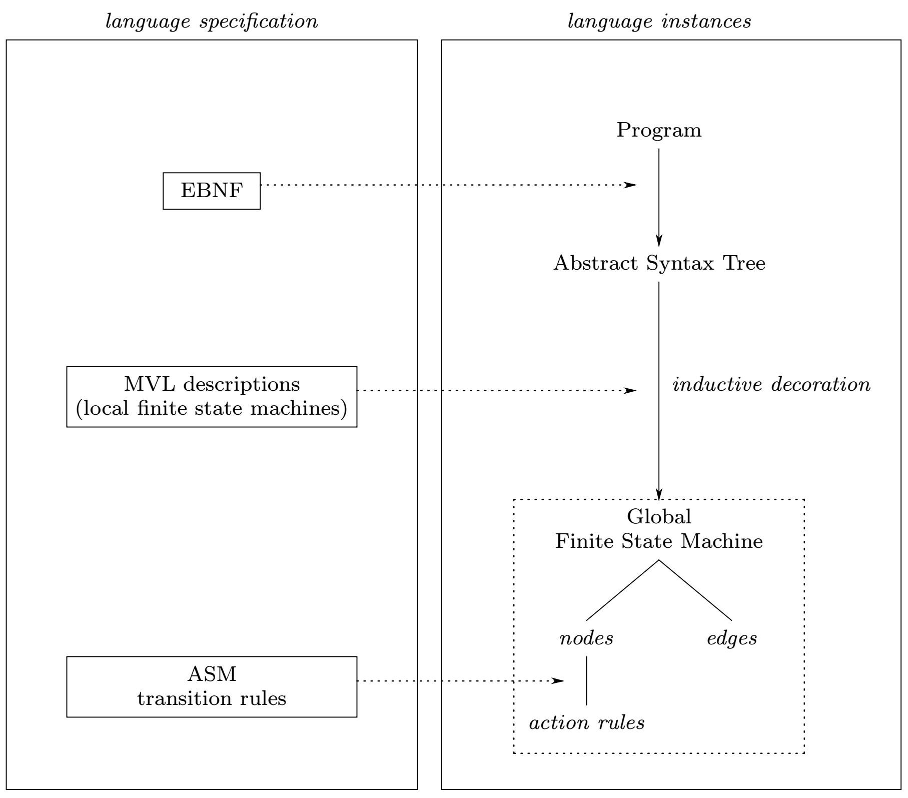


Fig. 1. Relationship between language specification and instances.

As an example the Montage for a nonterminal with name Sum is shown in Fig. 2. The topmost parts of this Montages is the production rule defining the context-free syntax. The remaining part defines static aspects of the construct given by means of an MVL description. Additionally, the Montage contains an action rule, which is evaluated after the two operands, i.e. when the control reaches the sum node.

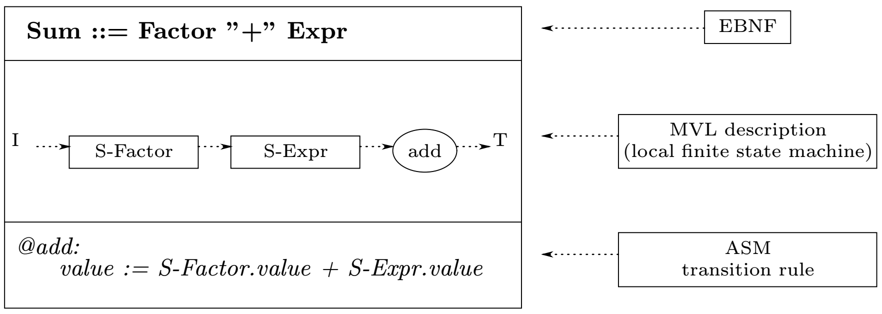


Fig. 2. Montage components.

The definition of Montages usually contains a fourth section which is devoted to the specification of the static semantics. As we are not using this property in the current paper, it will not be described. Future work will use this feature to formalize the elaborated static-semantics of AN, following the work in [Ørb94].

2.1 From Syntax to AST

In this section, the first step in Fig. 1 is described. As a result of this step we get the abstract syntax tree of the specified program. But we also compose the Montages corresponding to the different constructs of the language. This composition of the partial specifications is done based on the structure of the AST.

EBNF rules The syntax of the specified language is given by the collection of all EBNF rules. Without loss of generality, we assume that the rules are given in one of the two following forms:

$$A ::= B\ C\ D \tag{1}$$
$$E\ =\ F \mid G \mid H \tag{2}$$

The first form defines that A has the components B, C, and D whereas the second form defines that E is one of the alternatives F, G, or H. Rules of the first form are called *characteristic productions* and rules of the second form are called *synonym productions*. We guarantee that each non-terminal symbol appears in exactly one rule as the left-hand-side. Non-terminal symbols appearing on the left of the first form of rules are called *characteristic symbols* and those appearing on the left of synonym productions are called *synonym symbols*.

Composition of Montages Each characteristic symbol and certain terminal symbols define a *Montage*. A Montage is considered to be a *class*[2] whose instances are associated to the corresponding nodes in the abstract syntax tree. Symbols in the right-hand side of a characteristic EBNF rule are called *(direct) components* of the Montage, and symbols which are reachable as components of components are called *indirect components*. In order to access descendants of a given node in the abstract syntax tree, statically defined attributes are provided. Such attributes are called *selectors* and they are unambiguously defined by the EBNF rule. In the above given rule, the B, C, and D components of an A instance can be retrieved by the selectors S-B, S-C, and S-D. In Fig. 3 a possible representation of the A-Montage as class and an abstract syntax tree (AST) with two instances of A and their components are depicted.

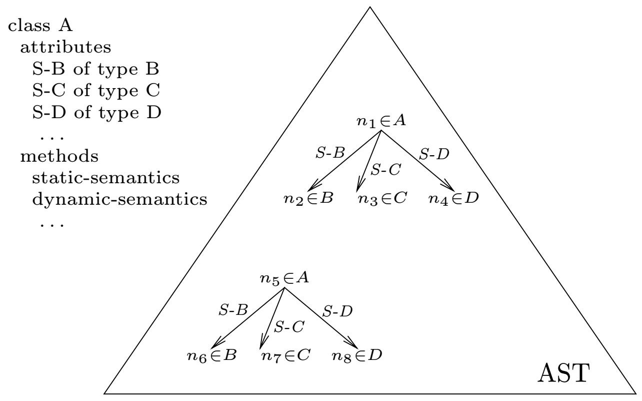


Fig. 3. Montage class A, instances in the AST, selectors S-B, S-C, S-D

Synonym rules introduce *synonym classes* and define subtype relations. The symbols on the right-hand-side of a synonym rule can be further synonym classes or Montage classes. Each class on the right-hand-side is a subtype of the introduced synonym class. Thus, each instance of one of the classes on the right-hand side is an instance of the synonym class on the left-hand-side, e.g. in the given example, all F-, G-, and H-instances are E-instances as well. In the AST, each

[2] In this context we consider class to be a special kind of abstract data type, having attributes and methods (actions) and, most important for us, where the notion of sub-typing and inheritance are predefined in the usual way.

inner node is an an instance of arbitrarily many (possibly zero) synonym classes and of exactly one Montage.

Terminals, e.g. identifiers or numbers, do not correspond to Montages. The micro-syntax can be accessed using an attribute *Name* from the corresponding leaf node. The described treatment of characteristic and synonym productions allows for an automatic generation of AST from the concrete syntax given by EBNF, see also the work in [Ode89].

Induced structures Inside a Montage class, the term *self* denotes the current instance of the class. Using the selectors, and knowledge about the AST, we can build paths w.r.t. to self. For instance, the path *self.S-B.S-H.S-J* denotes a node of class J, which can be reached by following the selectors S-B, S-H, and then S-J, see Fig. 4. The use of such a path in a Montage definition imposes a number of constraints on the other EBNF rules of the language. The example *self.S-B.S-H.S-J* requires that there is a B component in the Montage containing the path. Further, every subtype of B must have an H component, and every subtype of H must have an J component. In other words, the path *self.S-B.S-H.S-J* must exist in all possible ASTs.

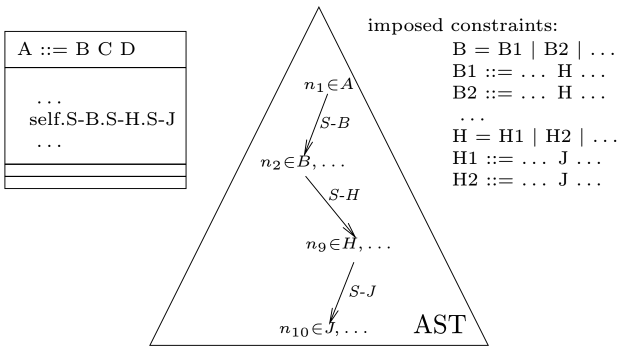


Fig. 4. Montage A using path self.S-B.S-H.S-J, situation in AST, and constraints on EBNF rules of B, H.

Example As a running example we give a small language $\mathcal{S}$. The expressions in this language potentially have side effects and must be evaluated from left to right. The atomic factors are integer constants and variables of type integer. The start symbol of the EBNF is Expr, and the remaining rules are

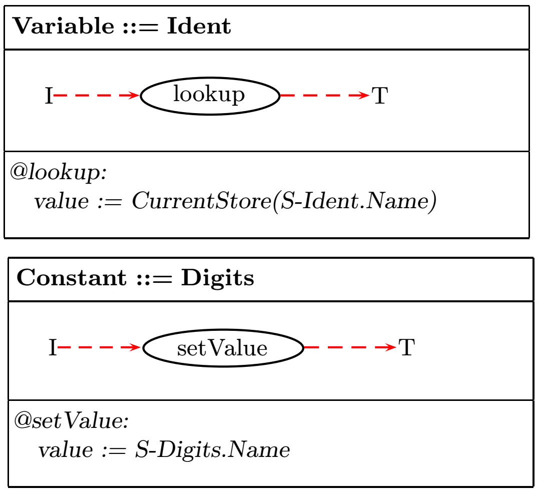
Expr	=	Sum | Factor
Sum	::=	Factor "+" Expr
Factor	=	Variable | Constant
Variable	::=	Ident

Constant ::= Digits

The following term is an $\mathcal{S}$-program:

```
2 + x + 1
```

As a result of the generation of the AST and the composition of the individual Montages shown in Fig. 2 and Fig. 5 we obtain the structure represented in Fig. 6.



@lookup:
  value := CurrentStore(S-Ident.Name)

@setValue:
  value := S-Digits.Name

Fig. 5. The Montages for the language $\mathcal{S}$.

In particular, the nodes from 1 to 8 represent instances of the Montage classes and the edges point to the successors of a particular node. The edges are labeled with the selector functions which can be used in the Montage corresponding to the source node to access the Montage corresponding to the target node. The nodes themselves show the class hierarchy starting from the synonym class and ending with the Montage class. The leaf nodes contain the definition of the attribute Name, i.e. the micro-syntax.

2.2 From AST to Control Flow Graphs

According to Fig. 1, the next step in building the data structure for the dynamic execution is the inductive decoration of the AST with a number of finite state machines. Again, this process is described rather informally here.

As we have seen in Fig. 2 and Fig. 5, the second part of a Montage contains the necessary specifications given in form of the *Montage Visual Language* (MVL). Two kinds of information are represented here: (a) the local state machine to be associated to the node of the AST and (b) information on the embedding of this

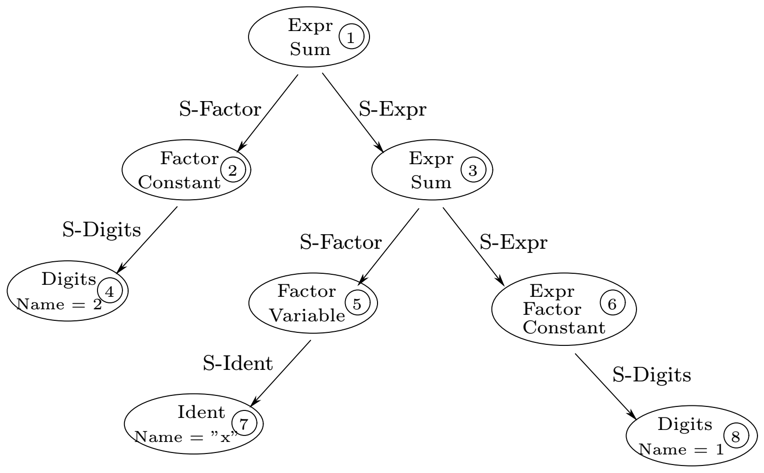


Fig. 6. The abstract syntax tree and composition of Montages for `2 + x + 1`

local state machine. Using our running example, Fig. 7 just represents the MVL sections of the Montages as they are associated to the corresponding nodes of the abstract syntax tree. The hierarchical state transition graph resulting from

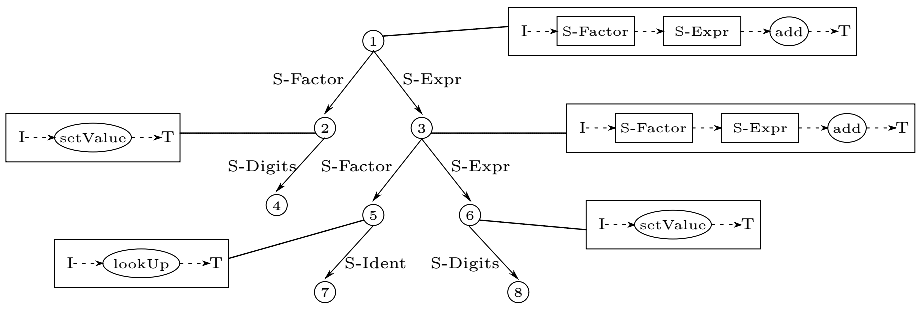


Fig. 7. The finite state machines belonging to the nodes.

the inductive decoration is shown in Fig. 8 for the running example.

Montage Visual Language Now, the elements of the MVL and their semantics can be described as follows:

– There are two kinds of nodes. The oval nodes represent states in the generated finite state machine. These states are associated to the AST node corresponding to the Montages. The oval nodes are labeled with an attribute. It

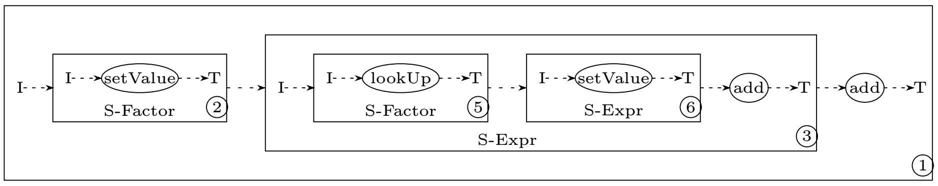


Fig. 8. The constructed hierarchical finite state machine.

serves to identify the state, for example if it is the target of a state transition or if it points to a dynamic action rule.

- The rectangular nodes or boxes represent symbols in the right hand side of the EBNF rule and are called direct components of a Montages, see Section 2.1. They are labeled with the corresponding selector function. Boxes may contain other boxes which represent indirect components. This way, paths in the AST are represented graphically.
- The dotted arrows are called control arrows. They correspond to edges in the hierarchical state transition graph of the generated finite state machine. Their source or target can be any box or oval. In addition, their source or target can be either the symbol I (I stands for initial) or T (T stands for terminal), respectively. In a Montage, at most one symbol of each, I and T, is allowed. If the I symbol is omitted, the states of the Montage can only be reached using a jump, if the T symbol is omitted, the Montages can only be left using a jump.
- As in other state machine formalisms (such as Harel's StateCharts), predicates can be associated to control arrows. They are simply terms in the underlying ASM formalism and are evaluated after executing the action rule associated to the source node. Predicates must not be associated to control arrows with source I.
- There are additional notations not used in this paper — for example data flow edges representing the mutual access of data between Montages and box structures representing lists in an effective way. Moreover, in this section of a Montage, one may specify further action rules to be performed in the static analysis phase, for example building up data structures necessary for the static and dynamic semantics.

It remains to show how the hierarchical finite state machine, for example Fig. 8 is built and how its dynamic semantic is defined.

Hierarchical FSM Building the hierarchical FSM is particularly simple. The boxes in the MVL are references to the corresponding local state transition graphs. Remember that nested boxes correspond to paths in the AST. Therefore, there are references to children only, i.e. to other state transition graphs along the edges of the AST. After resolving the references, a representation as in Fig. 8 is obtained.

Dynamic Semantics After the static analysis phase action rules are executed which define the dynamic semantics of the language.

- States of the finite state machines are visited sequentially.
- The action rule associated to a visited state is executed. The specification of these actions is based on the ASM formalism and specified in Section 2.3.
- The control is passed to the next state along a control arrow whose predicate evaluates to true. The control predicate, i.e. a term in the ASM formalism, is evaluated after executing the action associated to the source node.
  If there is more than one possible next state, the system behaves like a nondeterministic FSM. Up to now we did not use nondeterministic FSMs.
- If the target of a control arrow is a T, then a control arrow leaving the corresponding box in the enclosing parent state machine is followed. The term parent refers to the partial ordering of local state machines as imposed by the AST.
- If the target of a control arrow is a box, the corresponding local state machine corresponding to it is entered via the symbol I.

More formally, the arrows from I and to the T symbols define two unary functions, *Initial* and *Terminal* denoting for each node in the AST the first, respectively last state that is visited. According to the above description, the inductive definition of these functions is given as follows.

For each state s in the finite state machines,

$$s.\mathit{Initial} = s \tag{3}$$

$$s.\mathit{Terminal} = s \tag{4}$$

and for each instance n of a Montage N whose MVL-graph has an edge from I to a component denoted by path *tgt*,

$$n.\mathit{Initial} = n.\mathit{tgt}.\mathit{Initial}$$

and for each instance m of a Montage M whose MVL-graph has an edge from a component denoted by path *src* to T,

$$m.\mathit{Terminal} = m.\mathit{src}.\mathit{Terminal}$$

Using these definitions, the structured finite state machine can be flattened. The arrows of the flat finite state machine are given by the following equations defining the relation *ControlArrow*. For each instance n of a Montage N and each edge e in the MVL-graph of N,

$$\mathit{ControlArrow}(n.\mathit{src}.\mathit{Terminal}, n.\mathit{tgt}.\mathit{Initial})) = \mathit{true}$$

where *src* is the path of the source of e and *tgt* is the path of the target of e.

Applying these definitions to the running example results in the flat state machine of Fig. 9. In the same figure the dotted lines denote the relation of a state to its corresponding Montage, which is accessible as *self*. Using the Montages

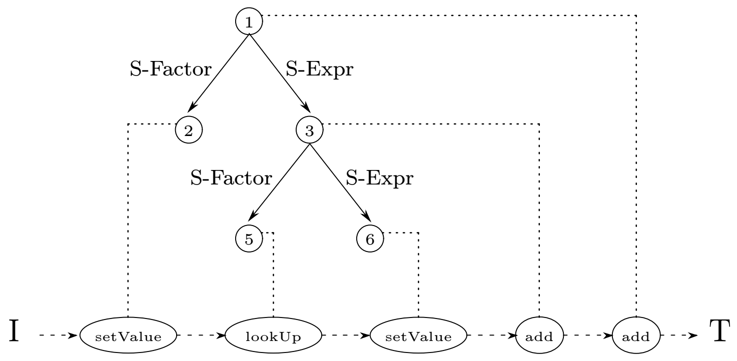


Fig. 9. The flat finite state machine and its relation to the AST.

shown in Figs. 2 and 5 and their action rules, we can track how the ASM rule associated with the *add* states can access the AST-nodes of its left and right arguments as *self.S-Factor* and *self.S-Expr.* The results of calculations performed by the actions are stored in the additional attributes *value.* The *add* action accesses the values of its arguments using the selectors, and defines its own *value* field to be the sum of the arguments. Assuming that *CurrentStore* maps x to 4, the execution of the flat or structured finite state machine sets the value of node two to the constant 2, sets the value of node five to the current store at x, sets the value of node six to 1, sets the value of node three to the sum of 4 and 1, and finally sets the value of node one to the sum of 2 and 5.

Please note that *all* the informally described concepts have been formalized using ASM notation. Even in the implementation, a Montages specification is at first transformed into static functions and ASM rules which are then executed by an ASM simulation engine. Therefore, the specification of control may also be provided in the dynamic semantics of a Montages. We use this possibility in the specification of AN in Section 3 via the function *JumpTo.* The underlying ASM formalism is described in the next section.

2.3 Dynamic Semantics by means of ASM rules

Basic ASM formalism The fundamental concept in ASMs is the *object*, which we consider an atomic entity. We call the set of all objects the *super-universe* $\mathfrak{U}$. In any ASM, $\mathfrak{U}$ contains the distinct objects *true, false* and *undef.* Additional examples for objects in the Montages context are the nodes of the abstract syntax tree, and the states of the finite state machines.

The *state* λ of the system is given as the mapping of a number of function symbols, the *signature* Σ, to actual functions. For short we write f_λ for the function interpreting the symbol f in state λ. We write $\mathfrak{S}(\Sigma)$ for the set of all Σ–states.

The functions interpreting the symbols need *not* be strict with respect to *undef*. In particular, the equality symbol is defined such that *undef* = *undef* evaluates to *true*. In our framework Σ contains a unary function for each attribute of a Montage. Examples are the selector-attributes, the attributes denoting the states, and specific to our running example the attribute *value*. Subsets[3] of $\mathfrak{U}$ are modeled by functions from $\mathfrak{U}$ to $\{true, false\}$. Such a function delivers *true* for all members of a set (instances of a type), and *false* otherwise. The set or type consisting of *true* and *false* is called *Boolean*. For each montage M, Σ contains the set M of M-instances. In the examples we have seen the sets *Expr*, *Factor*, *Sum*, *Variable*, *Constant*, *Ident*, and *Digits*. These sets can be used to characterize the types of functions, for instance

$$value : Sum \cup Variable \cup Constant \rightarrow Integer$$

A state transition changes these functions pointwise, by so-called *updates*. An update is a triple

$$(f, (o_1, \ldots, o_n), o_0)$$

where f is an n-ary function symbol in Σ, and $o_0, \ldots, o_n \in \mathfrak{U}$. Intuitively, *firing* this update at a state λ changes the function associated with f in λ at the point $(o_1, \ldots, o_n)$ to the value o_0, leaving the rest of the function unchanged.

Examples for rules are the actions *add*, *lookUp*, and *setValue* in Figs. 2 and 5. For instance the rule in Fig. 2

value := S-Factor.value + S-Expr.value

abbreviates

self.value := self.S-Factor.value + self.S-Expr.value

The meaning is that the attribute *value* of *self* is updated to the sum of the values of the left and right argument, which are accessed by means of the selector attributes *S-Factor* and *S-Expr*. Not only the semantics of the action rules, but the complete semantics of the finite state machine plus their construction is given by an ASM rule.

The formal semantics of a rule R in a state λ is given by a deterministic denotation *Upd(R, λ)* being a set α of updates [Gur97]. Given an update set α and a state λ, firing α at λ results in the successor state $\lambda' = \alpha(\lambda)$. We then have the following relation between f_λ and the new functions $f_{\lambda'}$:

$$f_{\lambda'}(o_1, \ldots, o_n) \longmapsto \begin{cases} o_0 & if \quad (f, (o_1, \ldots, o_n), o_0) \in \alpha \\ f_{\lambda'}(o_1, \ldots, o_n) & otherwise \end{cases} \tag{5}$$

[3] traditionally ASM literature speaks about universes

Update sets are defined by transition rules. The basic update denotes one update of a function at some point. The new values and the point are given by terms over the signature. Rules can be composed in a parallel fashion, such that all updates are executed at once. Conditional execution of a rule fires only in certain cases. The do-forall rule allows to fire the same rule for all objects satisfying some condition. Finally the extend-rule allows to introduce reserve-objects, that have not been used before. The last rule is typically used to create new objects of some class.

Formally, a transition rule R is built up recursively by the following constructions. The corresponding denotations are given. Let $eval_\lambda$ be the usual term evaluation over the state λ.

(Upd 1: basic update) if $R = \boxed{f(t_1, \ldots, t_n) := t_0}$
where $t_0, \ldots, t_n$ are terms over Σ,
then $Upd(R, \lambda) = \{(f, (eval_\lambda(t_1), \ldots, eval_\lambda(t_n)), eval_\lambda(t_0))\}$

(Upd 2: parallel composition) if $R = \boxed{R_1 \ldots R_m}$

then $Upd(R, \lambda) = \bigcup_{i \in \{1,\ldots,m\}} Upd(R_i, \lambda)$

(Upd 3: conditional rules) if $R = \boxed{\textbf{if } t \textbf{ then } R_{true} \textbf{ else } R_{false} \textbf{ endif}}$

then $Upd(R, \lambda) = \begin{cases} Upd(R_{true}, \lambda) & if \quad eval_\lambda(t) = true \\ Upd(R_{false}, \lambda) & otherwise \end{cases}$

(Upd 4: do forall) if $R = \boxed{\textbf{do forall } e : t \quad R' \textbf{ enddo}}$

then $Upd(R, \lambda) = \bigcup_{o : eval_{(\lambda \cup e \mapsto o)}(t)} Upd(R, \lambda \cup e \mapsto o)$

(Upd 5: extend) if $R = \boxed{\textbf{extend } E \textbf{ with } e \quad R' \textbf{ endextend}}$

then $Upd(R, \lambda) = Upd(R, \lambda \cup e \mapsto o)$,
where o is a completely new allocated element.[4]

In addition to the existing ASM concepts we use a number of structuring concepts well known from object oriented and functional programming. The formal semantics of these concepts are given in terms of basic ASMs.

Classes and Methods In Section 2.1 we introduced classes whose instances are the nodes in the AST. As already noted the instances of a class S are modeled by a universe S in the signature and attributes of the class are unary functions, whose domain are the instances of the class. In Section 3 this technique will be used to present several ADTs which encapsulate basic concepts of Action

[4] See [Gur97] for a formalization of “completely new”.

Notation. Experience showed that in this way, the semantics and tools support of Montages can be freely extended in new areas.

In addition, classes allow multiple inheritance, and recursive, dynamically bound methods. The sub-typing of classes and synonym classes mentioned in Section 2 is an application of inheritance. The method calls have a value parameter semantics, and are used at several places in Section 3.

Constructors The concept of terms built up by constructors can be mapped to the ASM approach as follows: each of the function names may be marked as *constructive*, expressing that constructor functions are 1-1 and total.

Let $\Sigma_c \subseteq \Sigma$ be the set of all constructive function symbols. If $f \in \Sigma_c$, with arity n, then the following conditions hold for all states A of the ASM:

(i) $\forall t_1, \ldots, t_n, eval_A(t_i) \neq undef, 1 \leq i \leq n \quad : \quad f(t_1, \ldots, t_n) \neq undef$
(ii) $\forall g \in \Sigma_c$, arity of g is m; $\; t_1, \ldots, t_n, eval_A(t_i) \neq undef \quad :$
$\quad f(t_1, \ldots, t_n) = g(s_1, \ldots, s_m) \Leftrightarrow$
$\quad f = g \wedge n = m$
$\quad \wedge eval_A(t_i) = eval_A(s_i), 1 \leq i \leq n$

where $eval_A(t)$ stands for the evaluation of term t in state A of the ASM. Informally speaking that means that each constructive function is (i) total with respect to $\mathfrak{U}$ and (ii) injective. If $f \in \Sigma_c$, then f is called a *constructor*, and the terms $f(t_1, \ldots, t_n)$ are called *constructor terms.* In the following, we use the constructor term t as a synonym for its unique value $eval_A(t)$.

For instance, the stack constructors *empty* and *push* can now be defined as follows:

```
constructor empty, push(_,_)
```

In addition, it is possible to define universes that are built up by constructor terms. For example, defining a universe *Stack* as

```
universe Stack = { empty, push(_,_) }
```

introduces the constructive functions *empty* and *push*, $eval_A(empty) \in Stack$ and $eval_A(push(t_1, t_2)) \in Stack$, for all terms t_1, t_2.

If a constructor definition is syntactically contained in a class definition C, then the constructor terms are put into the corresponding universe C. For example, the following constructor definitions are equivalent to the previous ones:

```
class Stack is
  constructors empty, push(_,_)
   ...
```

Pattern Matching Based on the concept of constructor terms, pattern matching functionality is provided. A *pattern matching equation* is a conditional term of the form $t_1 =\tilde{} t_2$. The *pattern term* t_2 may contain any numbers of *pattern variables* of the form "&x". This kind of equations perform the pattern matching operation well-known from the functional programming context.

Consider the following equational specification

$$\begin{aligned} x.push(y).pop &= x \\ x.push(y).top &= y \\ empty.top &= undef \\ empty.pop &= empty \end{aligned}$$

then the ASM translation is given by the following.

```
class Stack is
  constructor empty
  constructor push(_,_)

  method top is
    if self =~ push(&s, &d) then
      top_result := &d
    else
      top_result := undef

  method pop is
    if self =~ push(&s, &d) then
      pop_result := &s
    else
      pop_result := empty
```

In Section 3 the class Stack is used to simulate structure-based control flow concepts, which do not correspond to the finite state machine based model in Montages.

The ADT Stack exemplified a functional specification style which in turn can be freely mixed with the typical imperative ASM techniques based on updates of functions, as proposed in [Ode98]. In Section 3 many examples for this technique are shown, including a refinement from a functional specification of an ADT Map into a more imperative specification. In [Ode98] it is show how such refinements can be proved to be correct.

The tool support of Montages is based on the ASM compiler Aslan [Anl]. Aslan is a conservative and faithful implementation of ASM as defined by Gurevich in [Gur95] and [Gur97]. Furthermore Aslan presents some extensions, including classes and constructors, whose semantics has been formalized.

3 Action Notation Specification

In the Montages specification of the Action Notation-formalism (MAN) the exploitation of state-based features along with structure-based ones allows an architecture specification to reduce the interdependence between modules. This architecture follows and refines the partition of Action Notation (AN) into facets. In Section 3.1 an overview on this architecture is given. Section 3.2 shows how we implemented data notation(DN), and how terms are accessed and evaluated. The structure based features are illustrated in the description of the basic facet (Section 3.3). The full power of the state based features are shown in Section 3.4 where we give the specification of the main aspects of the imperative facet.

3.1 Architecture

The formal description of the AN consists of an interconnection of specification modules. Each module contains some local information, which possibly makes use of the behavior described in some other module. The specification architecture is illustrated in Fig. 10. The decomposition was done to separate the different

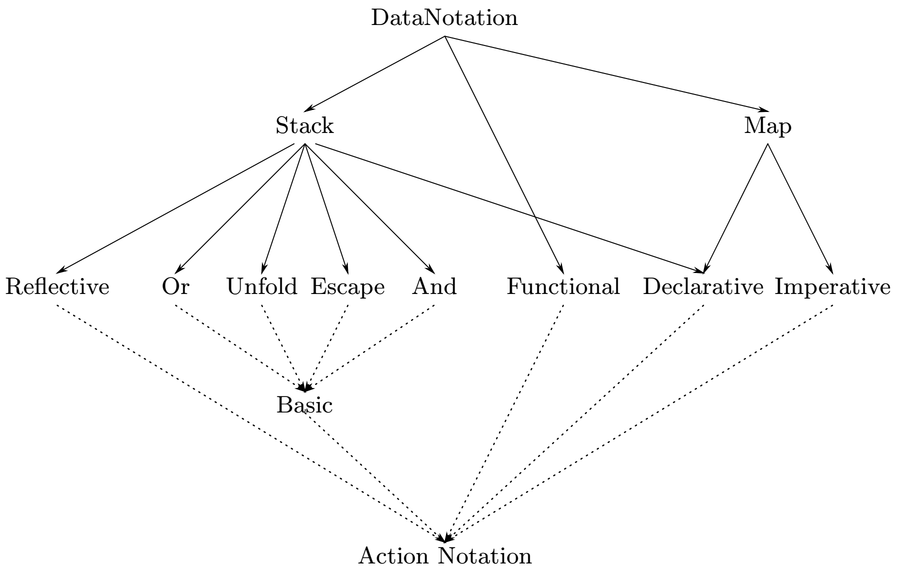


Fig. 10. Specification architecture

concerns involved in the design. Solid arrows in the graph denote import relation. The module *DataNotation*, described in Section 3.2, is used in all other modules.

The module *Stack* consists of the ADT Stack together with some functions described in Section 3.3. Stack is used both by the module *Reflective* and the four modules *Or*, *Unfold*, *Escape*, and *And* which are partitioning the actions of the basic facet into four modules. Each of these modules works in isolation, e.g. it refers to no part of other facets. In Section 3.3 the Or and Unfold modules are introduced in detail.

The module *Map* consists mainly of the definition of ADT Map. This ADT is used both in the *Imperative* facet and the *Declarative* facet. In Section 3.4 Map is informally introduced and used to explain the imperative facet. Later in Section 3.5 a simple and a refined specification of Map are shown.

The *Declarative* facet combines the techniques introduced in the description of the basic and the imperative facet. The declarative as well as the reflective facet are not further described in this text. The *communicative* facet is not yet included in our tool.

The described modules can be arbitrarily combined. The composition along the dotted arrows can always be reduced to expressions of the following form as depicted in figure 11.

$$M = M_1 \oplus_{M_0} M_2 \tag{6}$$

This means that the module M is obtained by the union of M_1 and M_2, which shares the module M_0. We call it, *composition* of M_1 and M_2 via M_0.[5]

$$\begin{tikzcd}
 & M_0 \arrow[dl] \arrow[dr] & \\
M_1 \arrow[dr, dotted] & & M_2 \arrow[dl, dotted] \\
 & M &
\end{tikzcd}$$

Fig. 11. A simple module decomposition.

For instance, if we consider the specification *Basic*, it is obtained by

$$\mathit{Basic} = \mathit{Or} \oplus_{\mathit{Stack}} \mathit{Unfold} \oplus_{\mathit{Stack}} \mathit{Escape} \oplus_{\mathit{Stack}} \mathit{And} \tag{7}$$

i.e. the composition of *Or*, *Unfold*, *Escape*, and *And* via *Stack*.

The complete specification of Action Notation is obtained by combining all modules of the architecture. Among the possible combinations of a smaller number of modules, we would like to mention the binary combination of the *functional* facet with one of the other modules. Using these combinations, useful examples of the single facets can be tested in isolation.

[5] In frameworks which have been categorically characterized, e.g. many-sorted algebraic specification, this operator usually corresponds to the *push-out* construction [EGRW98,EM85].

3.2 Data Notation and Yielders

Part of the AN is the Data Notation (DN), which is a collection of abstract data types given in terms of *algebraic specifications*. The predefined parts of DN can be implemented and executed, by interpreting the equations defining them as a term rewriting system. In MAN we include DN as a part of the underlying algebra. For each constructor in DN we define a constructor which is part of our ASM model. Formally the set *DN* contains all terms built up with these constructors.

The function *Eval* is used to evaluate data notation:

$$\mathit{Eval} : \mathit{DN} \rightarrow \mathfrak{U}$$

The concrete syntax rules of DN are included in the EBNF of MAN. The concrete syntax of DN-term t is parsed and transformed in the abstract syntax tree for t. The root of the tree (and of all subtrees) is in turn reconnected with the correct DN constructor term by means of the attribute

$$\mathit{data} : \mathit{DN\text{-}parse\text{-}tree} \rightarrow \mathit{DN}$$

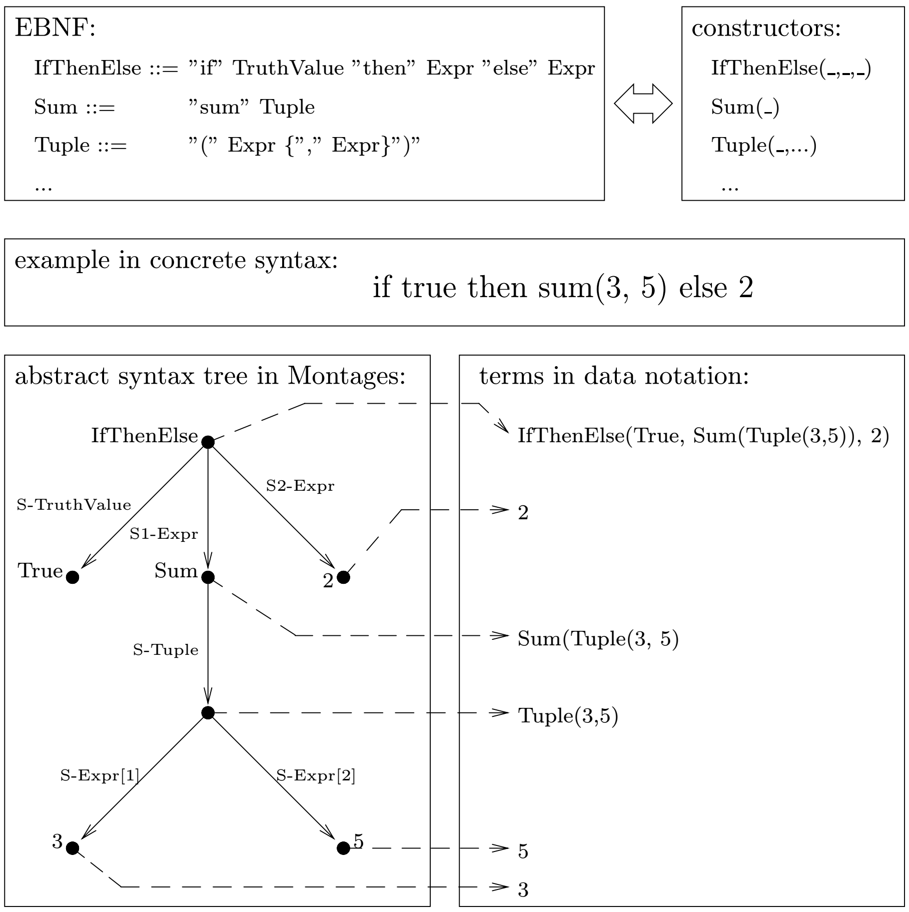
Fig. 12 illustrates this process. A part of the EBNF of DN is given together with the corresponding canonical definition of constructors. The term “if true then sum(3,5) else 2” is an example of the provided DN fragment in concrete syntax. The corresponding abstract syntax tree is sketched, and each node is related to the corresponding term in *DN*, by means of the attribute *data* which is depicted by dotted arrows.

So far *DN* is completely static and corresponds to algebraic specifications. So-called *yielders* extend DN with constructs whose evaluation depends from the state or current information. As an example we shall see in the next subsection yielders that depend on the current storage. The semantics of a yielder is given by defining how it is evaluated by means of *Eval*.

3.3 The Basic Facet

In the basic facet four different forms of control flow are supported. These forms differ considerably from the control flow induced by the finite state machines in Montages. The main difference is that these forms depend on the surrounding structure of the AST in a dynamically recursive way, while the FSM support of Montages determines a static next-state relation. Where no static next-state relation exists, the control flow has to be modeled by explicit jumps. To determine the jump targets, we introduce a stack simulating the recursion.[6] The current value of the stack is given by a 0-ary dynamic function ranging over Stack. Since we use that function to simulate a situation similar to the one in structural approaches, we call it structural control-flow stack (SCS).

[6] Unfortunately we realized too late that recursive ASM calls following the structure of the ASTs could have been used to express structural control flow more directly.



EBNF:
IfThenElse ::= "if" TruthValue "then" Expr "else" Expr
Sum ::= "sum" Tuple
Tuple ::= "(" Expr {"," Expr}")"
...
constructors:
IfThenElse(_,_,_)
Sum(_)
Tuple(_,...)
...
example in concrete syntax:
if true then sum(3, 5) else 2
abstract syntax tree in Montages:
IfThenElse
S-TruthValue
S1-Expr
S2-Expr
True
Sum
2
S-Tuple
S-Expr[1]
S-Expr[2]
3
5
terms in data notation:
IfThenElse(True, Sum(Tuple(3,5)), 2)
2
Sum(Tuple(3, 5)
Tuple(3,5)
5
3

Fig. 12. The mapping from DN trees to DN terms

$$SCS :\rightarrow Stack$$

The ADT Stack has been introduced in Section 2.

Current Information Several actions in the basic facet are manipulating the so-called *current information*. Current information has several components which are introduced in the facets. The problem is to allow a description of the actions in the basic facet which is orthogonal to the other facets. As a solution we use the functions

$$\begin{aligned} \mathit{GetInformation} &: \ \rightarrow \mathit{Information} \\ \mathit{SetInformation} &: \ \mathit{Information} \rightarrow \\ \mathit{CombineInformation} &: \ \mathit{Information}, \mathit{Information} \rightarrow \mathit{Information} \end{aligned}$$

GetInformation is returning the current information; *SetInformation(i)* sets the current information to i, and *CombineInformation*(i_1,i_2) returns the information resulting out of combining i_1 and i_2. The concrete definition of these functions are refined in each facet. In the basic facet, no type of information is introduced, and the three functions correspond to skip rules. In the functional facet the transient information is introduced and in the declarative facet the scoped information is introduced. The state of the current store as used in the imperative facet is considered to be stable information. AN is designed such that stable information is not manipulated by above operations.

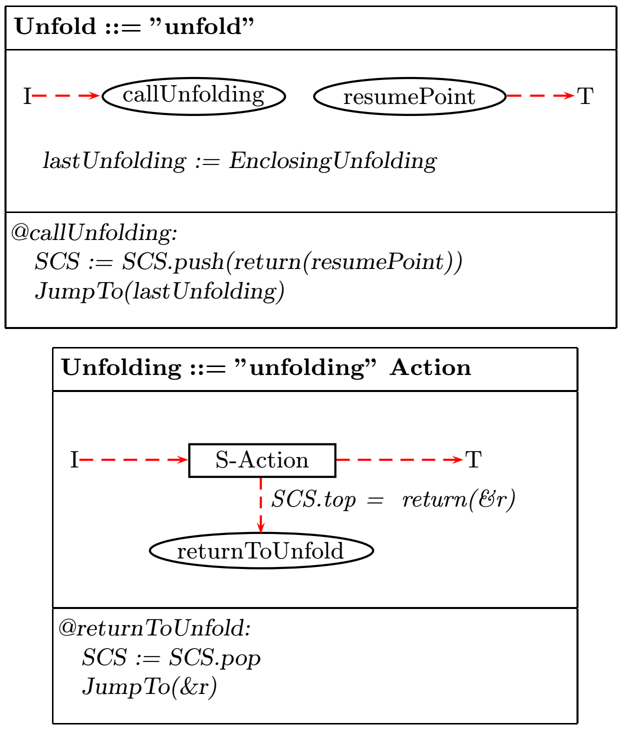
Unfolding and Unfold The composed action *Unfolding* and the primitive action *Unfold* are used mainly for iterative constructs. The meaning of *Unfold* is the execution of its syntactically least enclosing *Unfolding.* In the abstract syntax trees, the relation of Unfold instances to their least enclosing Unfolding instance is given by the lastUnfolding attribute.

$$\mathit{lastUnfolding} : \mathit{Unfold} \rightarrow \mathit{Unfolding}$$

In a functional AN model, one can regard *unfolding A* as an abbreviation for an action, generally infinite, formed by continually substituting *A* for *unfold.* In MAN we model *Unfold* as a call to the enclosing Unfolding. The *resumePoint* of the Unfold is put as return address on the stack, and control is passed to the lastUnfolding.

```
SCS := SCS.push(return(resumePoint))
JumpTo(lastUnfolding)
```

The Unfold montage in Fig. 13 thus has two action nodes, an unlabeled one, executing the above rule and the other serving as the resume point after a completed *Unfold.*
In the Unfolding montage in Fig. 13 first the Action component is executed. If after this execution there is a return Address $return(\&r)$ on the SCS, then control is passed back to &r, otherwise the Unfolding terminates.



```
@callUnfolding:
  SCS := SCS.push(return(resumePoint))
  JumpTo(lastUnfolding)
```

```
@returnToUnfold:
  SCS := SCS.pop
  JumpTo(&r)
```

Fig. 13. The Unfold and Unfolding Montages

Example 1. The following example calculates 2 to the power of 3. The action in the unfolding are two mutually exclusive alternatives which behave like an imperative *if-then-else* clause.

```
give 3 then
unfolding
( (check it is 0 then give 1)
or
  (check it is greater than 0 then
    give it - 1 then
    unfold then
    give it * 2
  )
)
```

And then The action *A1 and then A2* executes *A1*, *A2* sequentially. In contrast to Then (see Fig. 16) the current information before the execution is passed in parallel to both, *A1* and *A2*. The current information of *A1 and then A2* is obtained by combining the current information after the execution of *A1* with the current information after the execution of *A2*. As mentioned the stable information is not altered by the manipulations of current information.

The control graph of the AndThen montage is a simple sequence. The first action node *pushAction* pushes the current information on the stack. Then the control is passed to the first action, and then to the action node *swapAction*. The swapAction pops the information *&inf* from the stack, pushes the current information on the stack, and finally sets the current information to *&inf*.

Or, Fail, and Commit The action *A1 or A2* represents implementation-dependent choice between alternative actions. If *A1* or *A2* fails, the other alternative is tried.

The first action node of Or chooses nondeterministically *d* among *left* and *right*. If *d* is equal to

left then the alternative right branch *S2-Action.Initial* is pushed together with the current information as

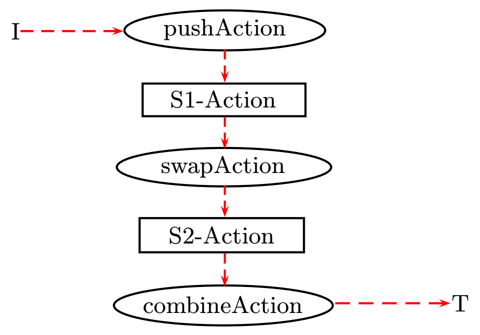
$$\textit{alternative}(\textit{S2-Action.Initial}, \textit{GetInformation})$$

where *alternative(_, _)* is a constructor. Further control is passed to the left branch *S1-Action.Initial*, using the *JumpTo(_)* operation which allows for non local and dynamically redirected arrows in the finite state machine.[7]

right the left branch is pushed, and control passed to the right one.

[7] These jumps can be defined graphically in MVL, but the needed graphical elements are not described here.

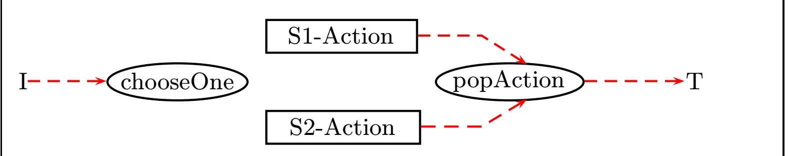
AndThen ::= Action ”and” ”then” Action



```
@pushAction:
  SCS := SCS.push(GetInformation)

@swapAction:
  if SCS = &l.push(&inf) then
     SCS := &l.push(GetInformation)
     !SetInformation(&inf)

@combineAction
  if SCS = &l.push(&inf) then
    SCS := &l
    !CombineInformation(&inf, GetInformation)
```

Fig. 14. The AndThen montage.

Or ::= Action "or" Action



```
@chooseOne:
  choose d in {left, right}
    if d = left then
      SCS :=
        SCS.push(alternative(S2-Action.Initial, GetInformation))
      JumpTo(S1-Action.Initial)
    else
      SCS :=
        SCS.push(alternative(S1-Action.Initial, GetInformation))
      JumpTo(S2-Action.Initial)

@popAction:
  SCS := SCS.pop
```

Fig. 15. The Or montage.

The primitive action Fail is modeled by an action node with the rule *!FailSemantics(SCS)*. The definition of FailSemantics is

```
method FailSemantics(x) is
  if  x =~ &s.push(alternative(&a, &i)) then
    !SetInformation(&i)
    SCS := &s.push(failed)
    JumpTo(&a)
  elseif x =~ &s.push(&any) then
    &s.!FailSemantics
  else
    _println("the program failed..")
```

In the case of an *alternative(&a, &i)* on top of the SCS, the current information is set to *&i*, *alternative(&a, &i)* is replaced with *failed*, and control is passed to *&a*. Otherwise *!FailSemantics* is called with the rest of the stack as argument. Like this the first alternative on the stack is searched recursively. If there is no alternative at all, the whole action failed, and execution is aborted.

The primitive action Commit replaces all *alternatives* on the stack with *failed* such that no backtracking is possible anymore.

3.4 The Imperative Facet

The imperative facet is the most natural part of AN to be specified with Montages because of the state-based nature of the framework. The simplest model of imperative behavior is a unary, dynamic function *CurrentStore* as used in the example of Section 2. A basic update *CurrentStore(x) := y* is used to update the store at position x to y. In the model of the imperative facet of AN, a more refined solution is needed.

In this section we give the specification of the imperative facet using an abstract data type *Map*. In contrast to the example in Section 2, the CurrentStore is a 0-ary function, ranging over the instances of Map.

$$\mathit{CurrentStore} :\to \mathit{Map}$$

The different operations are explained informally where needed, and in the Section 3.5 two alternative formal definitions of Map are given. One is a simple, abstract solution, and the other contains certain implementation-oriented decisions which overcome the problem of handling the store while dealing with large AN descriptions.

Reading the store The following yielder

YieldTheStoredAt ::= the Sort stored at Yielder is used to read the store at the cell denoted by the *Yielder*-component. If the result is of the specified sort, it is returned, otherwise the distinct element *nothing* is returned.

Using the definitions of selector attributes, the sort and yielder components can be accessed as *S-Sort* and *S-Yielder*, respectively.

For the integration in DN, a constructor *theStoredAt(_, _)* is introduced, and the field *data* is defined according to Section 3.2:

$$y.data = \mathit{theStoredAt}(y.\mathit{S\text{-}Store}.data, y.\mathit{S\text{-}Yielder}.data)$$

where y ranges over the instances of *YieldTheStoredAt*.

The definition of *Eval* is extended with the following equation:

```
Eval(theStoredAt($s$, $y$)) =
  let $r$ = CurrentStore.lookUp($y$.Eval) in
    if $r$ of type Eval($s$) then $r$ else *nothing*
```

where *_.lookUp(_)* is an operation of Map, used to read the store.

In AN one is only allowed to write and read in instances of the universe *Cell*. Cells are allocated using the primitive actions *Reserve*, *Unstore*, and *Unreserve*. For the ease of presentation, we do not use these, but instead we assume the existence of some yielders

$$\mathit{cell}_0, \mathit{cell}_1, \ldots$$

evaluating to instances of type *Cell*.

Getting a snapshot of the store A primitive yielder

YieldCurrentStorage ::= current storage

is introduced to get a snapshot of the store. The corresponding constructor is *currentStorage*, the definition of *data* is

$$y.data = \mathit{currentStorage}$$

where y ranges over the instances of *YieldCurrentStorage*. The extension of the definition of *Eval* is

Eval(currentStorage) = CurrentStore.getCopy

where *_.getCopy* is an operation of Map returning a snapshot of the current storage.

Writing the store The primitive action

StoreIn ::= store Yielder in Yielder

is used to write the datum *S1-Yielder.data* in the cell *S2-Yielder.data*.

The StoreIn-montage consists of an action node associated with the following transition rule.

CurrentStore.upDate(S2-Yielder.data.Eval, S1-Yielder.data.Eval)

where *_.upDate(_, _)* is an operation of Map used to update the store.

In the definition of the imperative facet we have seen that *Map* is an abstract data type having the following operations.

$$\begin{aligned} \mathit{upDate} &: \ \mathit{Map}, \mathit{Datum}, \mathit{Datum} \rightarrow \\ \mathit{lookUp} &: \ \mathit{Map}, \mathit{Datum} \rightarrow \mathit{Value} \\ \mathit{getCopy} &: \ \mathit{Map} \rightarrow \mathit{Map} \end{aligned}$$

The first operation has an imperative behavior, whereas the second and third operations have a functional behavior. In the next section we introduce a direct implementation of the described behavior, and then a refined version where both the first and the third operations use imperative techniques.

Example 2. The following example makes use of the yielder $cell_0$, as explained above.

```
store 1 in cell0 then
store 2 in cell0
```

First 1 is stored in $cell_0$, then the content of $cell_0$ is overwritten by 2.

Example 3. The following example illustrates the use of *YieldCurrentStorage*.

```
store 5 in cell0 then
store current storage in cell0 then
store current storage in cell0
```

This AN description is executed in three steps. After the first step the store maps $cell_0$ to 5. After the second step, $cell_0$ is mapped to a snapshot of the store after the first step. After the third step, the store maps $cell_0$ to a copy of the store after the first two steps.

Then action combinator In the presented examples we made use of the *Then* action combinator, although conceptually the then-action does not belong to the imperative facet. The definition of *Then* by means of Montages is given in Fig. 16. The control graph of the *Then*-montage defines graphically

- the left action (accessible with S1-Action) to be the *initial* in the control flow of *Then*

– the right action (accessible with S2-Action) to follow the left sequentially and
– the right action to be the *terminal* in the control flow of *Then*.

The *Then*-combinator does not alter any kind of information thus no additional action node is required for it. Applying this montage to the above example results in a simple sequence of two *StoreIn* actions.

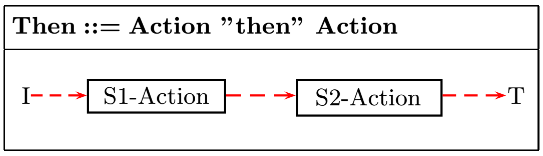


Fig. 16. The montage for the Then constructor

3.5 The ADT Map

The missing part for our model of the imperative facet is the definition of Map. We start by a simple definition that illustrates the use of ASMs. Later on, a more refined version is presented. The more refined version illustrates how our approach allows to solve efficiency bottle-necks on the specification level.

Simple Definition of Map Both for the simple and the refined definition, the ADT *Map* has an attribute *map*, being a unary, dynamic function from *Datum* to *Datum*. In the imperative facet this attribute is used to map cells to values.[8] In the object oriented style of our ASM interpreter the signature of *Map* is given as

```
class Map  is
  funattr map(_)
  method lookUp(x)
  method upDate(x,y)
  method getCopy
```

Initially the attribute *map* is undefined everywhere. No typing has to be provided, only the arity of attributes is needed. The *lookUp* and *upDate* operations are defined as:

```
 method lookUp(x) is
   lookUp_result:= map(x)
```

[8] In the declarative facet *map* is used to map tokens to bindable values.

```
method upDate(x,y) is
  map(x) := y
  upDate_result := self
```

The first operation is having a functional behavior, returning the result of reading the map-attribute, while the second method has an imperative behavior, having the side effect to update the map-attribute and returns the given instance of Map.

Consider the execution of *Example 1* starting with an empty store. After the first store action, *CurrentStore(*$cell_0$*)* is equal to 1. After the second store, *CurrentStore(*$cell_0$*)* is equal to 2.

The operation *getCopy* can be added to the simple definition as follows. A new Map is allocated and then the complete definition of the map is copied:

```
method getCopy is
  extend Map with newMap
    do forall x in dom map
      newMap.map(x) := map(x)
    enddo
    getCopy_result := newMap
```

This simple and abstract solution makes the tool unusable for large AN descriptions using the imperative facet. Thus in the next paragraph we propose an alternative refined definition of *Map*. The unusual property of this refinement is, that we mix functional and imperative specification parts.

Refined Definition of Map A more effective solution is to use a linked list of maps. The link from one map to the last is given by an attribute

$$\mathit{lastMap} : \mathit{Map} \rightarrow \mathit{Map}$$

To *lookUp* a value in such a list, all maps starting from the head are searched recursively. An *upDate* to a list of maps is performed on the head of the list only, the rest can be built up by snapshots. If a snapshot is taken using *getCopy*, the map is not cloned as above, but it is only marked by setting a flag *copyTaken.* When then next *upDate* operation is done, first a new map is allocated, and the *lastMap* attribute of the new map is set to the old map. Then the update is done only on the new map, guaranteeing that the old map is not altered and can be used as a valid snapshot.

The signature of Map extended with lastMap and copyTaken is

```
class Map  is
  funattr map(_)
  attr    lastMap
  relattr copyTaken

  method lookUp(x)
  method getCopy
  method upDate(x,y)
```

In the *lookUp* method, first the attribute *map* is searched, and then *lookUp* is called recursively on *lastMap*:

```
method lookUp(x) is
  if map(x) != undef then
    lookUp_result := map(x)
  elseif lastMap != undef then
    lookUp_result := lastMap.lookUp(x)
  else
    lookUp_result := undef
```

The attribute *copyTaken* is declared as a 0-ary relation which is initialized as *false*. It is used as a flag, to remember whether a copy has been taken. The method *upDate* is functionally the identity function, and as a side effect it sets the *copyTaken* flag:

```
method getCopy is
  copyTaken := true
  getCopy_result := self
```

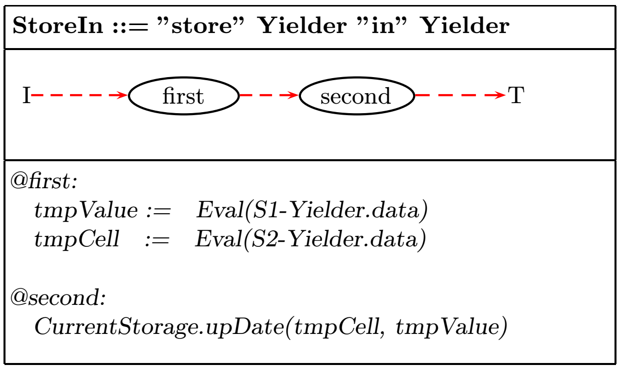
The update of such a map takes into account whether a copy has been taken. If yes, then a new map is allocated, and linked to the old one via *lastMap*. The new value is updated in the new map, and this map is returned.

If no copy has been taken, then no new map is needed and the update is made on the old map, which is returned as result.

```
  method upDate(x,y) is
   if not copyTaken then
     map(x):= y
     upDate_result := self
   else
     extend Map with m
       m.lastMap := self
       m.map(x) := y
       upDate_result := m
```

Assume we execute *Example 2* using the refined definition. At the end of the execution three instances of *Map* exist, reflecting the state of the store after step one, two and three. All of them are linked by *lastMap*.

A consequence of our proposed solution is that the evaluation of the components of StoreIn and the execution of upDate cannot be done simultaneously. In Fig. 17 the control flow of StoreIn consists thus of two action nodes, the first evaluating the components and storing the results in the attributes tmpValue and tmpCell, and the second executing the upDate operation.



StoreIn ::= "store" Yielder "in" Yielder

@first:
tmpValue := Eval(S1-Yielder.data)
tmpCell := Eval(S2-Yielder.data)

@second:
CurrentStorage.upDate(tmpCell, tmpValue)

Fig. 17. The Montage for the StoreIn action

4 The Generated Environment

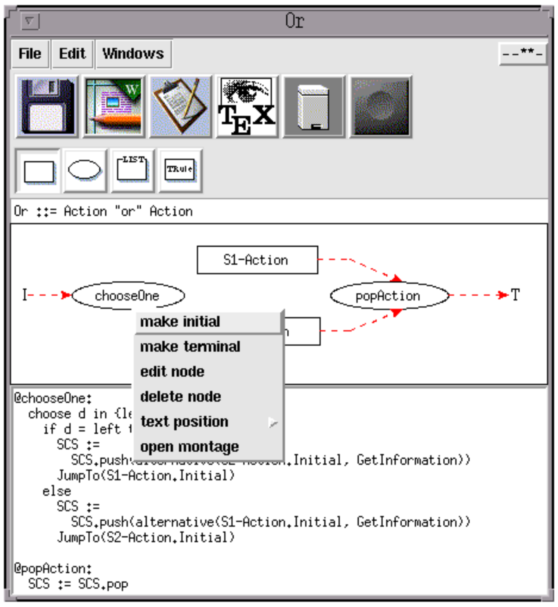
The development environment for Montages is given by the Gem-Mex tool [AKP97a,AKP97b]. It is a complex system which assists the designer in a number of activities related with the language design process. It consists of a number of interconnected components

– the Graphical Editor for Montages (Gem) is a sophisticated graphical editor in which Montages can be entered; furthermore documentation can be generated automatically; Fig. 18 shows the editor opened for the Or Montage;
– the Montages executable generator (Mex) which automatically generates correct and efficient implementations of the language;
– the generic animation and debugger tool visualizes the static and dynamic behavior of the specified language at a symbolic level; source programs written in the specified language and user-defined data structures can be animated and inspected in a visual environment.

In our case, the generated environment for AN has different purposes. In particular, while developing or extending the AN specification, it served to test empirically and validate the intended semantics of AN. At the same time, the same support can be used to visualize, debug, and explain the behavior of AN descriptions in terms of the specification (origin tracking).

4.1 Generation of Language Interpreters

Using the formal semantics description given by the set of Montages and the auxiliary ASM classes, the Gem-Mex system generates an interpreter for the specified language. No additional implementation details are requested to be provided by the designer. The core of the Gem-Mex system is *Aslan*, which stands for *A*bstract *S*tate Machine *Lan*guage and provides a fully-fledged implementation of the ASM approach. Aslan can also be used as a stand-alone,



```
Or ::= Action "or" Action
```

```
@chooseOne:
  choose d in {le
    if d = left t
      SCS :=
        SCS.push(                .Initial, GetInformation))
      JumpTo(S1-Action.Initial)
    else
      SCS :=
        SCS.push(alternative(S1-Action.Initial, GetInformation))
      JumpTo(S2-Action.Initial)

@popAction:
  SCS := SCS.pop
```

Fig. 18. The graphical editor of the Gem-Mex tool

general purpose ASM implementation. The process of generating an executable interpreter consists of two phases:

1. The Montages containing the language definition are transformed to an intermediate format and then translated to an ASM formalization (“`montages2asm`” in Figure 19).
2. The resulting ASM formalization is processed by the Aslan compiler generating an executable version of the formalization, which represents an interpreter implementing the formal semantics description of the specified language.

Using Aslan as the core of the Gem-Mex system provides the user the possibility to exploit the full power of the ASM framework to enrich the graphical ASM macros provided by Montages with additional formalization code.

4.2 Generation of Visual Programming Environments

Besides pure language interpreters, the Gem-Mex system is able to generate visual programming environments for the generated ASM formalization of the programming language semantics.[9] This is done by providing a generic debugging

[9] This feature is again available to all kind of ASM formalizations implemented in Aslan not only to those generated from a Montages language specification.

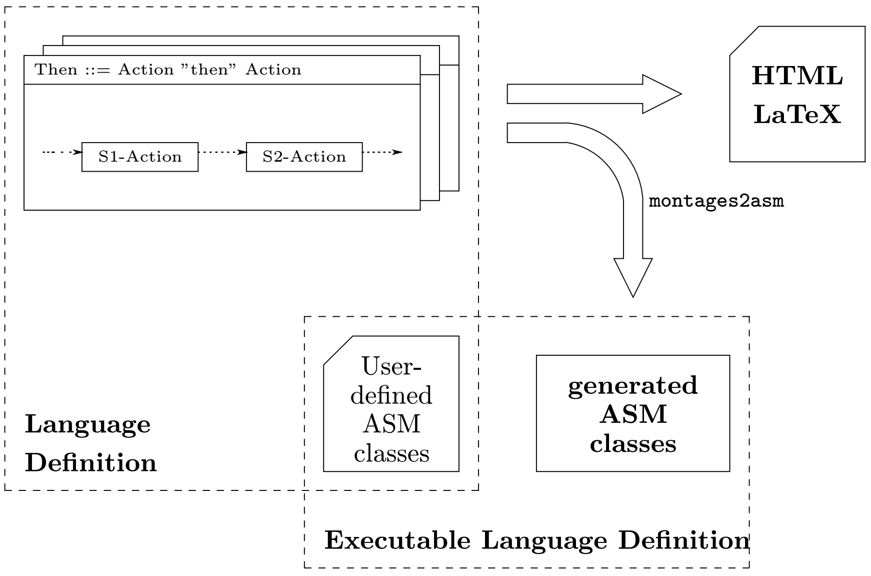
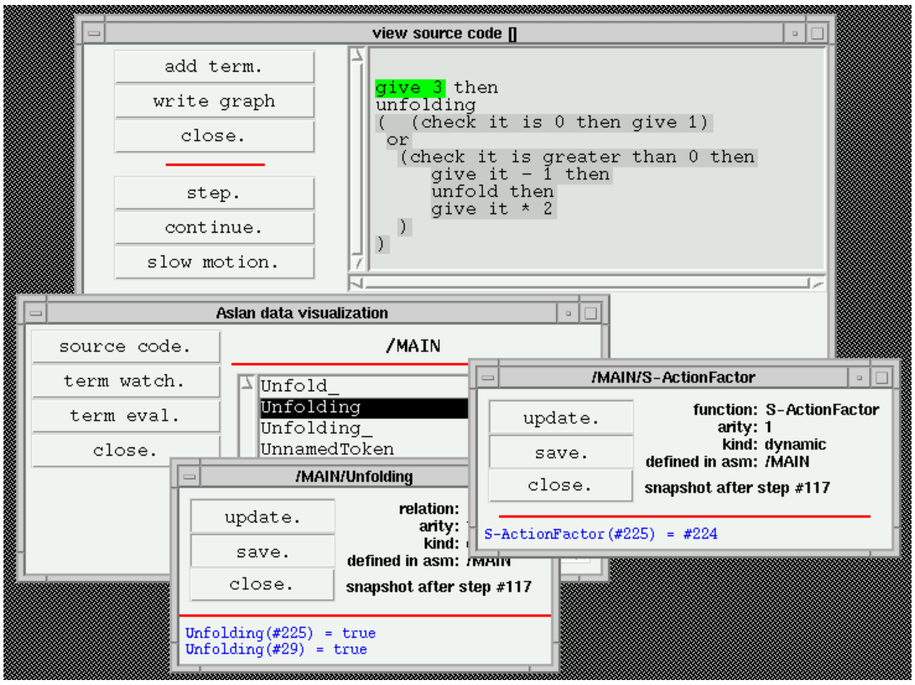


Fig. 19. The architecture of the Gem-Mex system

and animation component which can be accessed by the generated executable. During the translation process of the Montages/ASM code special instructions are inserted that provide the information being necessary to visualize the execution of the formalization. In particular, the visual environment can be used to debug the specification, animate the execution of it, and generate documents representing snapshots of the visualization of data structures during the execution. The debugging features include stepwise execution, textual representation of ASM data structures, definition of break points, interactive term evaluation, and re-play of executions. Fig. 20 shows an example of this kind of visualization, where the AN description of Example 1 is illustrated in the topmost window (*view source code*). In particular, for that particular description the universe *Unfolding* is containing the node denoted by #225[10] (window *Unfolding*) and the selector function *S-ActionFactor* links the unfolding node occurrence with its action argument (node #224), which is highlighted in the text (window *S-ActionFactor*).

Figure 21 shows an example of the graphical animation facility of the Gem-Mex system. On the right-hand-side of the window the AN program of Example 1 is visualized and the position information generated during the compilation process of the Montages is displayed. This position information is used, for example, to highlight certain parts of the source code that correspond to values of data structures contained in the language formalization. In Figure 21, the change of the value of the "current-task" function *CT* is animated by drawing

[10] The node #29 corresponds to the Aslan class definition of *Unfolding*.



```
give 3 then
unfolding
(  (check it is 0 then give 1)
 or
  (check it is greater than 0 then
     give it - 1 then
     unfold then
     give it * 2
  )
)
```

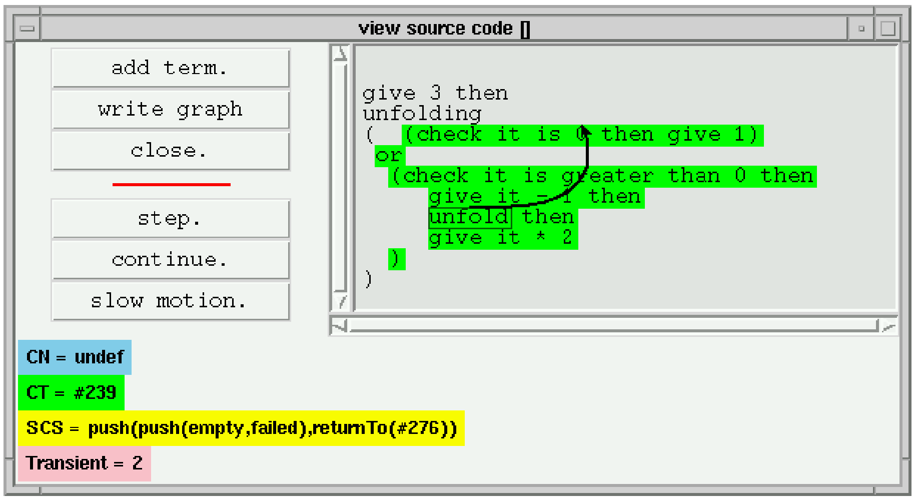
```
Unfold_
Unfolding
Unfolding_
UnnamedToken
```

```
function: S-ActionFactor
   arity: 1
    kind: dynamic
defined in asm: /MAIN
snapshot after step #117

S-ActionFactor(#225) = #224
```

```
      relation:
         arity:
          kind:
defined in asm: /MAIN
snapshot after step #117

Unfolding(#225) = true
Unfolding(#29) = true
```

Fig. 20. Textual Visualization of data structures in the Gem-Mex system

an arrow from *unfold* to its "replacement" given by the argument of *unfolding*. Experiences show that especially this kind of animation is useful to explain and document the formal semantics as specified in the Montages.



```
give 3 then
unfolding
(   (check it is 0 then give 1)
 or
  (check it is greater than 0 then
     give it - 1 then
     unfold then
     give it * 2
  )
)
```

```
CN = undef
CT = #239
SCS = push(push(empty,failed),returnTo(#276))
Transient = 2
```

Fig. 21. Graphical animation in the Gem-Mex system

With the "write graph" button in figure 21 one can trigger the production of a graphical representation of the syntax nodes and their interconnections, like data and control flow arrows, selection functions, and initial and terminal arrows. Gem-Mex generates an input file for the "VCG" tool [San95] which can be used to visualize these data structures. As an example, Figure 22 displays a portion of the abstract syntax tree of the examples program displayed in Figure 21.

4.3 Generation of Documentation Frames

As sketched in Figure 19 the Gem-Mex system also generates files that can be used as frames for the documentation of the language specification. Both paper and online presentation of the language specification are automatically generated:

- LaTeX documents illustrate the Montages and the grammar; such documents are easily customizable for the non-specialist user; all Montages in this paper are generated by Gem-Mex;
- HTML versions of the language specification allows to browse the specification and retrieve pieces of specification.

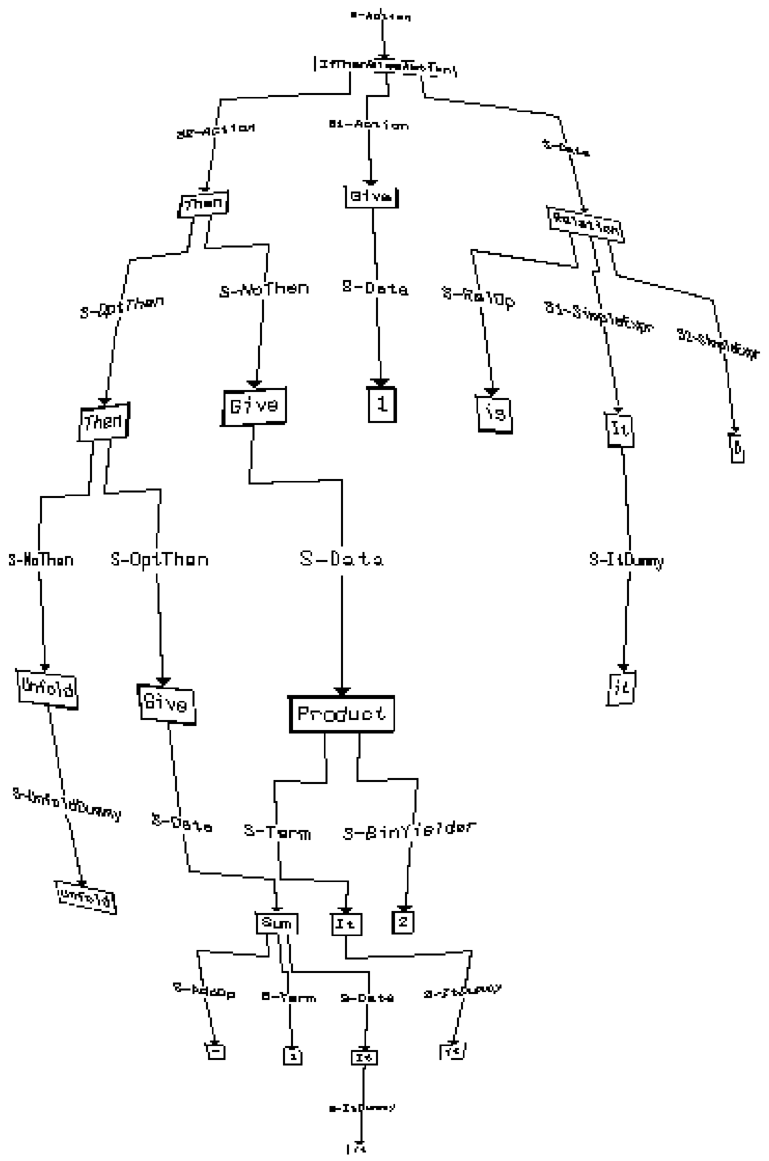


Fig. 22. Visualization of a portion of the abstract syntax tree of the example in Figure 21

4.4 Library of Programming Language Features

A concept for providing libraries of programming language features is currently under development. With this concept is shall be possible to reuse features of programming languages that have already been specified in other Montages. Examples for this kind of features are arithmetic expressions, recursive function call, exception handling, parameter passing techniques, standard control features etc. The designer of a new language can then import such a feature and customize it according to his or her needs. The customization may range from the substitution of keywords up to the selection among a set of variants for a certain feature, like different kinds of inheritance in object-oriented languages, for example. This would allow, for instance, to embed the AN behavior in other languages reusing part of it or extending it. In the Verifix project [HLT98], a number of reusable Montages has been defined with the intention to reuse not only the Montages but as well an associated construction scheme for correct compilers.

Acknowledgments We thank Samarjit Chakraborty, Christoph Denzler, Yuri Gurevich, Wuwei Shen, and Chuck Wallace for their collaboration in the Montages project.

References

[AKP97a] M Anlauff, P. W. Kutter, and A. Pierantonio. Formal Aspects of and Development Environments for Montages. In M. Sellink, editor, *2nd International Workshop on the Theory and Practice of Algebraic Specifications*, Workshops in Computing, Amsterdam, 1997. Springer.

[AKP97b] M. Anlauff, P. W. Kutter, and A. Pierantonio. The Gem-Mex Tool Homepage. `http://www.first.gmd.de/~ma/gem/`, 1997.

[AKP97c] M. Anlauff, P. W. Kutter, and A. Pierantonio. The Montages Project Web Page. http://www.tik.ee.ethz.ch/~montages, 1997.

[AKP98] M. Anlauff, P. Kutter, and A. Pierantonio. Enhanced Control and Data Flow Graphs in Montages. 1998. submitted for publication.

[Anl] M. Anlauff. The Aslan Language Manual. Part of the Aslan distribution.

[BH98] E. Börger and J. Huggins. Abstract state machines 1988 – 1998: Commented ASM bibliography. In H. Ehrig, editor, *EATCS Bulletin, Formal Specification Column*, number 64, pages 105 – 127. EATCS, February 1998.

[BW99] D. Brown and D. A. Watt. JAS: a Java Action Semantics. In *Proceedings of AS'99 (to appear)*, BRICS notes series, 1999.

[EGRW98] H. Ehrig, M. Große-Rhode, and U. Wolter. Applications of category theory to the area of algebraic specification in computer science. *APCS (Applied Categorical Structures)*, (6):1–35, 1998.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*, volume 6 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Berlin, 1985.

[GH93] Y. Gurevich and J. K. Huggins. *The Semantics of the C Programming Language*, volume 702 of *LNCS*, pages 274–308. Springer, 1993.

[Gur88] Y. Gurevich. Logic and the Challenge of Computer Science. In E. Börger, editor, *Theory and Practice of Software Engineering*, pages 1–57. CS Press, 1988.

[Gur95] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*. Oxford University Press, 1995.

[Gur97] Y. Gurevich. May 1997 Draft of the ASM Guide. Technical Report CSE-TR-336-97, University of Michigan EECS Department Technical Report, 1997.

[HLT98] A Heberle, W. Löwe, and M. Trapp. Safe reuse of source to intermediate language compilations. Fast Abstract, 9th International Symposium on Software Reliability Engineering, September 1998. http://chillarege.com/issre/fastabstracts/98417.html.

[Hug] J. Huggins. Abstract State Machines Web Page . http://www.eecs.umich.edu/gasm.

[KH95] P. W. Kutter and F. Haussmann. Dynamic Semantics of the Programming Language Oberon. Term work, ETH Zürich, July 1995. A revised version appeared as technical report of Institut TIK, ETH, number 27, 1997.

[KP97a] P. W. Kutter and A. Pierantonio. Montages: Specifications of Realistic Programming Languages. *JUCS, Springer*, 3(5):416–442, 1997.

[KP97b] P. W. Kutter and A. Pierantonio. The Formal Specification of Oberon. *JUCS, Springer*, 3(5):443–503, 1997.

[KST98] P. W. Kutter, D. Schweizer, and L. Thiele. Integrating Formal Domain-Specific Language Design in the Software Life Cycle. In *Current Trends in Applied Formal Methods*, LNCS. Springer, October 1998.

[MM93] M. A. Musicante and P. D. Mosses. Communicative action notation with shared storage. Tech. Mono. PB-452, Dept. of CS, Univ. of Aarhus, 1993.

[Mos92] P. D. Mosses. *Action Semantics.* Number 26 in Cambridge Tracts in theoretical Computer Science. Cambridge University Press, 1992.

[Mos98a] P. D. Mosses. Modularity in natural semantics (extended abstract). Available at http://www.brics.dk/∼pdm, 1998.

[Mos98b] P. D. Mosses. Modularity in structural operational semantics (extended abstract). Available at http://www.brics.dk/∼pdm, 1998.

[Ode89] M. Odersky. *A New Approach to Formal Language Definition and its Application to Oberon.* PhD thesis, ETH Zürich, 1989.

[Ode98] M. Odersky. Programming with variable functions. In *International Conference on Functional Programming*, Baltimore, 1998. ACM.

[Ørb94] P. Ørbæk. OASIS: An optimizing action–based compiler generator. In *CC'94, Proc. 5th Intl. Conf. on Compiler Construction, Edingurgh*, volume 786 of *LNCS*, pages 1–15. Springer Verlag, 1994.

[RW92] M. Reiser and N. Wirth. *Programming in Oberon - Steps Beyond Pascal and Modula.* Addison-Wesley, 1992.

[San95] G. Sanders. Graph layout through the vcg tool. In I. G. Tollis R. Tamassia, editor, *Graph Drawing, DIMACS International Workshop GD'94, Proceedings*, volume 894 of *Lecture Notes in Computer Science*, pages 194–205. Springer Verlag, 1995.

[vDM96] A. van Deursen and P. D. Mosses. ASD: The action semantic description tools. In Springer, editor, *AMAST'96 Proc., 5th Intl. Conf. on Algebraic Methodology and Software Technology*, number 1101 in LNCS, pages 579 – 582, Munich, 1996.

[Wal97] C. Wallace. The Semantics of the Java Programming Language: Preliminary Version. Technical Report CSE-TR-355-97, University of Michigan EECS Department Technical Report, 1997.

[Wan97] K. Wansbrough. A modular monadic action semantics. Master's thesis, Dept. of CS, Univ. of Auckland, February 1997.

[Wat91] D. A. Watt. *Programming Language Syntax and Semantics.* Prentice-Hall, 1991.

[Wat99] D. A. Watt. The Static and Dynamic Semantics of SML. In *Proceedings of the AS'99 (to appear)*, BRICS notes series, 1999.

[WG92] N. Wirth and J. Gutknecht. *Project Oberon, The Design of an Operating System and Compiler.* Addison-Wesley, 1992.

JAS: a Java[⋆] Action Semantics

Deryck F. Brown[1] and David A. Watt[2]

[1] School of Computer and Math Sciences, The Robert Gordon University, St Andrew Street, Aberdeen AB25 1HG, Scotland. db@scms.rgu.ac.uk
[2] Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, Scotland. daw@dcs.gla.ac.uk

Abstract. A formal specification of Java is badly needed. The current Java language specification (Gosling et al., 1996), as is typical of informal specifications, is inconsistent, incomplete and inaccurate.
The aim of the JAS project is to produce a complete formal specification of the Java language, its virtual machine, and selected parts of its API, all using action semantics. The first stage in the project is to produce a formal specification of the dynamic semantics of the Java language.
The formal specification of the Java language has produced some interesting insights into the design of the language, and into using action semantics to specify object-oriented constructs.
This paper describes the use of action semantics in the specification of the Java programming language, and shows how feedback from the specification process has identified several issues that are not covered in the existing informal specification.

1 Introduction

The aim of the JAS project is to produce a complete formal specification of the Java language, the Java Virtual Machine, and the core Java classes, all using action semantics. These specifications will be developed over a number of stages. This paper discusses the first of these stages: the production of a formal specification of the dynamic semantics of the Java language.

Currently, the definitive specification of the Java language is the Java Language Specification (JLS) (Gosling et al., 1996). However, as is typical of informal specifications, it is inconsistent, incomplete and inaccurate. Some of the problems identified with the JLS are listed in the Java Spec Report (Perara & Bertelsen, 1998).

This paper reports the work to date on the specification of the Java language using action semantics. Section 3 discusses the current draft of our specification of the dynamic semantics of the Java language. Like most action semantic descriptions, this specification consists of three main parts: abstract syntax, semantic functions, and semantic entities. Here, we concentrate on the semantics entities of Java, which illustrates how action semantics can be used to specify object-oriented features. The abstract syntax is largely based on the syntax given

[⋆] Java is a trademark of Sun Microsystems Inc.

in the JLS, and preserves most of the naming convention used therein. Most of the semantic functions share features of other programming languages, and are therefore less specific to Java. To illustrate the object-oriented features of Java, we only consider the semantic function for method invocation here.

The specification of the Java language has not been without its problems. Section 4 discusses the problems encountered with action notation, and Section 5 discusses problems encountered with the JLS and the Java language. Others have already studied how the Java language might be formally specified (Börger & Schlute, 1998; Wallace, 1997), and have identified some language problems using other formalisms. However, the use of action semantics to study the Java language is unique to the JAS project.

Finally, Section 6 discusses how the current specification will be extended to include the concurrent features of Java, and the features added in version 1.1 of the language.

The current (draft) version of the action semantics of Java can be found at `http://www.dcs.gla.ac.uk/~daw/publications/JAS.ps`.

2 Background

Java is ubiquitous and needs little introduction in this paper. Several points, however, do need to be made.

First, Java is much more than a programming language. Java consists of a programming language, a virtual machine implementation, and an ever increasing number of APIs (or class libraries). This paper relates only to the Java programming language.

Second, Sun Microsystems intends to standardize Java, including the Java language. It is their intention to "fast-track" the standardization process, and avoid the delays associated with a standardization committee[1]. This means that any defects or anomalies in the Java language need to be identified quickly, and resolved, before it becomes a standard. We argue that the production of a complete, formal specification of the Java language is vital in addressing these issues.

Third, there are a number of proposals for extending and enhancing the Java language (e.g. by adding parametric polymorphism). These enhancements must be judged by their impact on the Java language. A good way to judge their impact is by extending the formal specification to include these new features.

3 Java dynamic semantics

Our current action semantics of the Java language attempts to formalise the description given in the JLS. As such, it covers version 1.0 of the Java language.

[1] Recent announcements from Sun indicate that they are not going to submit Java as an ISO standard using their ability to submit a Java specification directly as a standards document. This would require them to hand maintenance of the Java standard to ISO as well, which they are unwilling to do.

In this initial version, however, the concurrent features of the Java language are not addressed. The addition of these features is discussed in Section 6.1.

In the following sections, we discuss the object-oriented features of Java. We consider the representation of classes, fields, methods, constructors, interfaces, and objects. We then consider the semantics of method invocation.

3.1 Classes

A *class declaration* introduces a new reference type, which denotes a class. A class declaration contains the following items:

- a *class name*. This name, together with the current package name, gives the fully qualified name of the class, which uniquely identifies it.
- an optional *direct superclass*. A class declaration may extend the definition of an existing class. The extended class becomes the *direct superclass* of the new class. If the superclass is not specified explicitly, then the class declaration extends the class `java.lang.Object`. Only the class `java.lang.Object` has no (direct) superclass.
- a (possibly empty) list of *interfaces*. A class declaration may implement a number of existing interfaces. These interfaces become the *direct superinterfaces* of the new class. A class may have no direct superinterfaces (but it may have indirect superinterfaces if its superclass implements any interfaces).
- a *class body*. The class body may contain *class member* (i.e. field and method) declarations, *constructor* declarations, and *static initializers*.

A class declaration may contain one of the modifiers `public`, `abstract` or `final`. Of these modifiers, only `abstract` has a dynamic effect in the sense that an abstract class may contain abstract method declarations, which have no dynamic semantics.

A *constructor* is used in the creation of an object that is an instance of a class. A constructor is invoked normally by a class instance creation expression (using the operator `new`). A class may contain several constructors that are distinguished by their signatures.

A *static initializer* is a block of statements, and a class may contain several such initializers. Any static initializers of a class are executed when the class is *initialized*, they (together with any *field initializers*) may be used to initialize the values of any class variables belonging to the class. A class is initialized on its first *active use*, i.e. when either a method or constructor in the class is called, or when a non-constant field declared in the class is used or assigned.

A class member declaration introduces either a new variable (field) or a new method. Such a declaration can contain a number of *modifiers*. Most modifiers are used to control access to the member, and are not relevant in the dynamic semantics. The only modifier with a dynamic effect that we will consider here is `static`[2]. Class members (fields and methods) fall into the following four categories:

[2] Other modifiers such as `transient`, `synchronized`, and `volatile`, which also have a dynamic effect, are not handled in the current specification.

- *class fields* or *variables.* A class variable is introduced using a `static` field declaration, and it is instantiated once regardless of the number of instances of the class that exist.
- *instance fields* or *variables.* An instance variable is introduced using a non-`static` field declaration, and it is instantiated in each instance of the class that exists.
- *class methods.* A class method is introduced using a `static` method declaration, and it is invoked without reference to a particular object (i.e. it cannot access any instance members of the class).
- *instance methods.* An instance method is introduced using a non-`static` method declaration, and it is invoked with reference to a particular object, which is denoted by `this`.

Fields and methods are considered in more detail in Sections 3.2 and 3.3.

A class is defined by the following sort in action semantics:

$$\begin{aligned}\mathsf{class} = \mathsf{class\ of}\ (&\mathsf{type\text{-}name\text{-}token},\\ &\mathsf{field\text{-}bindings},\ \mathsf{field\text{-}allocator},\ \mathsf{type\text{-}initializer\text{-}cell},\\ &\mathsf{class\text{-}method\text{-}bindings},\ \mathsf{instance\text{-}method\text{-}bindings},\\ &\mathsf{constructor\text{-}bindings},\ \mathsf{interface\text{-}bindings},\ \mathsf{class}^{?})\ .\end{aligned}$$

The various components of this sort are as follows:

- The type-name-token specifies the fully-qualified name of the class.
- The field-bindings are bindings for the class variables declared in the class.
- The field-allocator is an abstraction that, when enacted, allocates storage for the instance variables of the class and initializes them to their appropriate values. It produces a set of bindings for the instance variables that are held as part of an object of this class.
- The type-initializer-cell contains an abstraction that, when enacted, initializes the class variables of the class. It represents the statements contained in static initializers and `static` field initializers. When a class has been initialized, the type-initializer is replaced by a harmless abstraction. This prevents the class from being initialized more than once.
- The class-method-bindings are bindings for the `static` method declarations in the class.
- The instance-method-bindings are bindings for the non-`abstract`, non-`static` method declarations in the class.
- The constructor-bindings are bindings for the constructor declarations in the class.
- The interface-bindings are bindings for the various (direct) superinterfaces of the class.
- The class is the direct superclass of this class. It is empty only for the class `java.lang.Object`.

3.2 Fields

A field declaration introduces a new variable. A `static` field declaration introduces a class variable, and a non-`static` field declaration introduces an instance variable.

A field declaration containing the `final` modifier, and whose initial value is known at compile-time, may be denoted by a binding of the declared identifier to a value, rather than to a variable initialized to contain that value. Since a `final` field, therefore, may be represented by a binding to either a variable or a value, we must remember which it is. We do this by associating a truth-value with each field, which is set to true if the field is `final`. This is required when using a `final` field, since it must always yield a value when accessed and never a variable.

This leads to the following sort definitions:

- field = field of (truth-value, variable | value) .
- field-bindings = map[token to field] .
- field-allocator = abstraction [giving field-bindings | storing | escaping]
  [using current storage] .

A field-allocator can be enacted to produce a set of field-bindings that contains a freshly allocated collection of instance variables, which have been initialized. The enactment of the abstraction may escape as a result of the initialization code throwing either an exception, such as `ArithmeticException`, or an error, such as `IllegalAccessError`.

3.3 Methods

A method declaration introduces a new method. A `static` method declaration introduces a class method, and a non-`static` method declaration introduces an instance method. An `abstract` method declaration has no dynamic semantics and is ignored.

A class may contain a number of overloaded method declarations that are distinguished by their signatures. The current specification does not, however, cover overloading.

Class methods and instance methods differ since a class method has no object associated with it, whereas an instance method does. The sorts class-method and instance-method therefore differ in the transients they receive. A class method is given a tuple of values for the arguments, and an instance method is given both the receiver object and the values for the arguments. Apart from this, both kinds of methods are represented similarly.

This leads to the following sort definitions:

- class-method-bindings = map[token to class-method] .
- instance-method-bindings = map[token to instance-method] .

- method-bindings = map[token to method] .
- class-method =
  abstraction [giving a value$^?$ | storing | diverging | escaping]
  [using the given value* | current storage] .
- instance-method =
  abstraction [giving a value$^?$ | storing | diverging | escaping]
  [using the given (object, value*) | current storage] .
- method = class-method | instance-method .

The result type of both kinds of method may be `void`, and so the corresponding abstraction sorts give an optional value. The enactment of the abstraction representing a method may escape as a result of the method body throwing an exception or an error.

3.4 Constructors

A constructor is used in the creation of an object that is the instance of a class. In most respects, a constructor looks like an instance method with no return value. It receives the newly allocated object, and the actual parameters supplied to it.

A class may contain a number of overloaded constructor declarations that are distinguished by their signatures. The current specification does not, however, cover overloading.

If a class contains no constructor declarations, then a default constructor, which takes no arguments, is automatically provided. For a class that has a direct superclass, this default constructor simply invokes the superclass constructor with no arguments.

This leads to the following sort definitions:

- constructor =
  abstraction [storing | diverging | escaping]
  [using the given (object, value*) | current storage] .
- constructor-bindings = map[token to constructor] .

The constructor itself is not responsible for the allocation of the object. This is the responsibility of the `new` operator. If the constructor was responsible for the allocation of an object, then this would complicate the construction of an object of a subclass, as the constructor would have to call the constructor for the superclass, which would then allocate an object of the superclass.

Like methods above, the enactment of the abstraction representing a constructor may escape as a result of the constructor body throwing an exception or an error.

3.5 Interfaces

An interface declaration introduces a reference type, which denotes an interface. An interface is similar to a class, but contains only `final`, `static` field (i.e. constant) declarations, and `abstract` method declarations. An interface may also extend a number of superinterfaces.

This leads to the following sort definition:

> interface = interface of (type-name-token, field-bindings,
> type-initializer-cell, interface-bindings) .

The representation of an interface is therefore just a reduced form of class. It contains the fully-qualified name of the interface, the bindings for the class variables (which are all constants), a type-initializer (to initialize the interface), and the bindings for the superinterfaces.

3.6 Objects

An *object* is an instance of a class, and so an object is a member of a particular class. An object contains its own copy of the instance variables of the class, which are allocated and initialized when the object is created. Objects have unique identities that can be compared to determine if two references are to the same object, or to different objects. Finally, the class of an object is used to determine the method body that is invoked by a method call, and to access any class variables.

A variable does not contain an object itself, but instead holds a reference to an object. There is a special reference, called `null`, that does not refer to any object.

This leads to the following sort definitions:

- reference = null | object (*disjoint*) .
- null : reference .
- object = object of (class, field-bindings, identity) .
- identity = cell .

An object contains a class, a set of field-bindings, and an identity.

The class denotes the class of the object. It contains the instance methods associated with this object, and information about this object's superclass and superinterfaces.

The field-bindings of the object contain all of the instance variables for this object, including the instance variables declared in the superclasses of this object's class. It is the concatenation of the collections of field-bindings produced by enacting the field-allocators of the object's class and all of its superclasses. The field-bindings may include fields that are *hidden* in the current object, but which will become visible if the object undergoes a narrowing conversion to a superclass. Since the field-bindings are a map, they cannot contain more than one binding to the same token. Thus the tokens representing the names of the instance variables must be distinguished, even for hidden fields. This is a similar problem to distinguishing the names of overloaded methods and constructors, and is not handled in the current draft specification.

The identity of an object is just a cell. This makes it easy to allocate new object identities, and to compare them for equality. The contents of the cell are never used.

3.7 Method invocation

In this section, we consider the most important object-oriented feature, namely method invocation. The JLS requires 17 pages to describe the semantics of method invocation, although this also includes overloading resolution, which is not considered here. Method invocation has six possible cases, since a method I may represent either an instance method or a class method, and it may be selected using a name, N (which ends in I), an expression, $E.I$, or the superclass, super.I.

In general, a method will return a result, and so a method invocation is evaluated to give a value. The corresponding semantic function is "evaluate _", which has the following definition:

```
• evaluate _ :: Method-Call →
        action [giving a value$^?$ | storing | diverging | escaping]
            [using current bindings | current storage] .
```

The first semantic equation is concerned with the case where a method is selected by a name (N):

```
(1)  evaluate ⟦ $N$:Name "(" $A$:Arguments$^?$ ")" ⟧ =
        | give the method and receiver denoted by $N$ and then
        | respectively evaluate $A$
        then
        | enact the application of the given class-method#1
        |     to the rest of the given (class-method, value$^*$) or
        | enact the application of the given instance-method#1
        |     to the rest of the given (instance-method, object, value$^*$) or
        | escape with the given throw#1 .
```

This action is structured as follows. First the name N is analyzed to give the corresponding method and receiver object, using the auxiliary operation "the method and receiver denoted by _". If the given method is a class-method, then there is no receiver object, otherwise the method will be an instance-method, and the receiver will be the object identified by N. Next the arguments are evaluated to produce a tuple of values. Finally, either the corresponding class method or instance method is enacted with the given arguments and receiver (where appropriate). If the evaluation of the arguments throws an exception, then no method is invoked, and the action escapes with the same exception.

The second semantic equation is concerned with the case where a method is selected by an expression ($E.I$):

```
(2)  evaluate ⟦ E:Expression "." I:Identifier "(" A:Arguments? ")" ⟧ =
         | evaluate E and then
         | respectively evaluate A
       then
         | | enact the application of the class-method I of the type of E
         | |       to the rest of the given (object, value*)
         | or
         | | check there is the instance-method I of the type of E and then
         | | | | enact the application of the instance-method I of the class of
         | | | |       the given object#1 to the given (object, value*)
         | | | or
         | | | | check the given value#1 is the null-reference then
         | | | | escape with the throw of the null-pointer-exception .
```

This equation is structurally similar to the first case. However, in this case, the expression E is evaluated to produce the receiver object, and the method selected is based on the *type* of the expression E (which may be different from the class of the given object). If the evaluation of the expression E gives a null-reference, and the method is an instance method, then no method is invoked, and the action escapes with a null-pointer-exception. However, if the selected method is a class-method then it is invoked, even if the evaluation of the expression E gives null.

The third semantic equation is concerned with the case where a method is selected by the superclass (super.I):

```
(3)  evaluate ⟦ "super" "." I:Identifier "(" A:Arguments? ")" ⟧ =
         | give the super-object of the object bound to this-token and then
         | respectively evaluate A
       then
         | enact the application of the class-method I of the given object#1
         |       to the rest of the given (object, value*) or
         | enact the application of the instance-method I of the given object#1
         |       to the given (object, value*) .
```

In this final case, the receiver object is identified by narrowing the current object ("the object bound to this-token") to an instance of its superclass ("the super-object of ..."). The class of the resulting object is then searched for the corresponding method, and either a class-method or an instance-method is enacted, in a similar way to the previous equations.

Here we have concisely and accurately specified the various forms of method invocation present in the Java language. The current specification, however, does not concern itself with method overloading (i.e. multiple methods with the same name and distinguished by their signatures). This will complicate the semantics of method invocation. Most of the detail, however, will be hidden inside auxiliary operations such as "the method and receiver denoted by".

4 Specification problems

There are several features of the Java language that are awkward to formally specify due to difficulties in action notation.

One problem relates to the scope of bindings in Java. Two or more class declarations can make mutually-recursive references to one another. Furthermore the member (field and method) declarations within these classes can also be mutually-recursive. For example, the following class declarations illustrate this problem:

```
class P {
  public static int a = Q.c; }

class Q {
  public static int c = 2;
  public static int d = P.a;
}
```

The class P declares a class variable a that refers to the class variable Q.c for its initialization. Next, the class Q declares the class variable c required by P.a, and a further class variable d that is initialized to the value of P.a.

In action semantics, recursive bindings are modeled using *indirections*. An indirect binding can be initially set to unknown, and later redirected to the correct bound value. This copes well with the "shallow" recursive bindings in languages such as Pascal. However, in Java, we have "deep" recursive bindings. In the above example, the class declaration for P depends not only on the existence of the class declaration for Q, but it can also look inside the class bound to Q and refer to the member declarations it contains. It is not sufficient to create an indirect binding for the class Q, and later replace it with the corresponding class value.

Using the indirect bindings of action semantics to specify this behaviour is error-prone, unclear, and unsatisfactory, even for experts in the notation. This is one area where action semantics fails to produce a *readable* specification.

Other problems identified with action notation relate to the specification of the concurrent features of Java, and are discussed in Section 6.1.

5 Java language problems

The production of the action semantics of Java has revealed a number of inconsistencies and omissions in the JLS. One such omission concerns the scope of declarations inside a switch-statement. The following example illustrates the problem:

```
switch (x) {
  case 1: int y = 6; ...; break;
  case 2: y = 3; break;
}
```

The body of the switch-statement is a block, and accordingly the scope of the variable declaration "`int y`" is from the point of declaration until the end of the block. Therefore, in the second case of the switch-statement, the variable `y` is still in scope. This means that all cases prior to the one selected must be processed, and any declarations they contain must be elaborated. However, such declarations are not initialized. So, in the above example, if the second case attempts to print the value of `y` without assigning to it, then the class fails to compile with the error "Variable `y` may not have been initialized".

This problem with the switch-statement is frequently overlooked. Börger and Schlute (1998) omit the switch-statement from their core Java subset, and Wallace (1997) incorrectly specifies the meaning of the switch-statement as the meaning of the selected case.

A second problem relates to the design of Java. An array assignment in Java has the general form:

$$E_1[E_2] = E_3;$$

As defined in the JLS, the semantics of the array assignment statement is non-compositional, i.e. the meaning of this statement is not composed from the meaning of its sub-phrases, namely a variable-access and an expression. Instead, all three expressions are evaluated before the array variable is identified. This means that, for example, any side-effects of evaluating E_3 will take place even if the value of E_1 is null and results in a `NullPointerException` being thrown. These side-effects also occur if the value of E_2 is too large, and results in an `ArrayIndexOutOfBoundsException` being thrown.

Finally, as noted in the JLS, the array assignment statement may also require a run-time type check to ensure that the result of the expression E_3 can be assigned to the variable denoted by $E_1[E_2]$. If the type check fails, an `ArrayStoreException` is thrown.

6 Extending the specification

6.1 Adding concurrency

Adding the concurrent features of Java will require a significant amount of work. A Java program may contain many *threads* that are instances of the class java.lang.Thread (or one of its subclasses), and execute in parallel. Threads are synchronized using the `synchronized` modifier, which allows a particular thread to lock a given object (or class) before executing a `synchronized` instance (or class) method. The lock prevents other threads from executing any `synchronized` methods of the object (or class) until the lock is released. Threads run in a shared memory space, and can see all of the variables in the entire program.

There are two main difficulties associated with specifying Java threads in action notation, and both are associated with the design of action notation itself.

The specification of threads will require the use of the communicative facet in action notation. Each thread will be represented by a separate agent that is contracted to perform the body of the thread. Unfortunately in action notation, communicative agents are executed with their own storage, and do not provide a simple mechanism for shared storage. This leads to our first problem: shared storage will have to be specified as a separate agent that maintains the store for all threads, and messages will have to be sent between agents to handle store accesses and updates.

The introduction of the communicative facet to represent essentially imperative behaviour has an unfortunate side-effect on the semantic description, and leads to our second problem. Currently, the semantic functions use the imperative facet to hold the values of variables, and they access the store using imperative yielders. However, if the imperative facet is replaced by the sending and receiving of messages in the communicative facet, then this would alter the style of the semantic functions. The communicative facet uses communicative actions to send and receive messages, so terms which are currently imperative yielders (e.g. "the _ stored in _") must be replaced by communicative actions. This has a significant impact on the style of the semantic functions, especially as several auxiliary functions are defined as yielders, and will now be actions instead.

6.2 Adding Java 1.1 features

The most significant addition in Java 1.1 is the introduction of *inner classes*. This allows class and interface declarations to be nested inside another class. However, since in the action semantics a class is bound to its fully-qualified name, and an inner class has a uniquely determined name, based on the concatenation of its name with the class it is inside, we do not expect the introduction of inner classes in the semantics to cause undue difficulty.

Java 1.1 also introduces an *object initializer*, which is analogous to a static initializer for a class, and consists of a block of instructions to initialize the instance variables of the class. The object initializer is executed immediately after the superclass constructor, but before the constructor of the class. Thus instance variables of the superclass are always initialized (either in the constructor or by an object initializer) before the instance variables of the class itself. Again, we do expect object initializers to cause undue complications.

The remaining additions in Java 1.1 are in the APIs supported (such as reflection, object serialization, remote method invocation, and beans). The JAS project as a whole will need to address these changes in due course. However, none of the changes have a direct bearing on the specification of the dynamic semantics of the Java language.

7 Conclusion

The current specification of the Java language is incomplete, but has already provided useful feedback on the design and informal specification of the Java language. It is anticipated that as the action semantics is completed, even more issues will be discovered.

Up until now, the JAS project has deliberately ignored other attempts at the formal specification of Java. This was to prevent misconceptions from being propagated, and to assess the success of action semantics at detecting problems in the existing language specification provided by Sun. Once the specification is completed, we will start to compare our action semantic description with other specifications, and attempt to discover any further inconsistencies and anomalies. A rich source of this material will be found in the recent volume on the syntax and semantics of Java (Alves-Foss, 1998).

Finally, another attempt at an action semantics of the Java language has come to our attention (Diaz, 1999). This specification is also currently in an incomplete form, but it will be interesting to compare it directly with the specification produced by the JAS project.

References

Alves-Foss, J. (Ed.). (1998). *Formal syntax and semantics of Java.* Lecture Notes in Computer Science, 1523. Springer-Verlag. (In press)

Börger, E., & Schlute, W. (1998). *A programmer friendly modular definition of the semantics of Java.* In Alves-Foss (1998).

Diaz, R. (1999). *Java action semantics.* Unpublished manuscript.

Gosling, J., Joy, B., & Steele, G. (1996). *The Java language specification.* Addison-Wesley.

Perara, R., & Bertelsen, P. (1998). *The Java spec report.* (`http://www.dina.kvl.dk/~jsr/`)

Wallace, C. (1997). *The semantics of the Java programming language: Preliminary version* (Tech. Rep. No. CSE-TR-355-97). University of Michigan, Department of Electrical Engineering and Computer Science.

Bootstrapped Semantics-Directed Compiler Generation

Stephan Diehl

FB 14 - Informatik, Universität des Saarlandes,
Postfach 15 11 50, 66041 Saarbrücken, GERMANY
`diehl@cs.uni-sb.de, http://www.cs.uni-sb.de/~diehl`

Abstract. We introduce our natural semantics-directed generator 2BIG for compilers and abstract machines. It applies a sequence of transformations to a set of natural semantics rules including a pass separation transformation. Then we discuss how it can be used to generate a compiler and abstract machine for action notation. With the help of these components we can then generate compilers for other source languages whose semantics has been specified in Action Notation. We also briefly discuss the concept of an abstract machine language based on the abstract machine generated for action notation.

1 Introduction

Given a semantics specification of a source language, current semantics-directed compiler generators produce compilers from the source language into a fixed target language.

$$\begin{tikzcd}
\text{semantics} \arrow[d] & \text{program} \arrow[d] & \text{input} \arrow[d] & \\
\text{generator} \arrow[r] & \text{compiler} \arrow[r] & \text{code} \arrow[r] & \text{output}
\end{tikzcd}$$

Rather than just generating compilers which translate source programs into a fixed target language, our system generates both a compiler and an abstract machine. The generated compiler translates source programs into code for the abstract machine.

$$\begin{tikzcd}
\text{semantics} \arrow[d] & \text{program} \arrow[d] & & \\
\text{generator} \arrow[r] \arrow[rrdd, bend right] & \text{compiler} \arrow[r] & \text{abstract machine code} \arrow[dd] & \\
& & & \text{input} \arrow[dl] \\
& & \text{abstract machine} \arrow[r] & \text{output}
\end{tikzcd}$$

We chose Action Notation[8] as an example of a realistic programming language, because it offers a rich set of primitives underlying both imperative and functional programming languages. Since Action Notation is used to write Action Semantics specifications, we can then combine the generated compiler for Action Notation with an Action Semantics specification of a programming language. As a result, we get a compiler from the programming language to the generated abstract machine language for Action Notation.

$$\begin{tikzcd}
\text{source program} \arrow[d] & & & \\
\text{Action Semantics of source language} \arrow[r] & \text{action term} \arrow[dd] & & \\
\text{semantics of Action Notation} \arrow[d] & & & \\
\text{generator} \arrow[r] \arrow[rrdd] & \text{compiler} \arrow[r] & \text{abstract machine code} \arrow[dd] & \\
& & & \text{input} \arrow[dl] \\
& & \text{abstract machine} \arrow[r] & \text{output}
\end{tikzcd}$$

Thus, in general, we can bootstrap semantics-directed compiler generators (SDCG): Given an implementation of an SDCG for a semantics formalism F_1, we can get an SDCG for an semantics formalism F_2, if we have a specification of F_2 in F_1.

2 Action Semantics

Action semantics [8] has been developed to allow for useful semantics descriptions of realistic programming languages. The language used to write such semantics descriptions is called *action notation*.

The semantic entities of action semantics are *actions*, *data* and *yielders*. Actions are computational entities, they reflect the step-wise execution of programs. Data are mathematical entities like numbers, truth-values, lists and sets. Finally yielders represent unevaluated data. If the action containing a yielder is performed, the yielder evaluates to a concrete datum. Actions can become data by encapsulating them in *abstractions*, which can be *enacted* into actions again. The performance of an action may *complete* (i.e., normal termination), *escape* (i.e., exceptional termination which may be trapped), *fail* (i.e., abandoning the performance of an action which can lead to the performance of an alternative action) or *diverge* (i.e., nontermination). Actions process different kinds of information and can be classified according to which *facet* they belong: *basic* (control-flow, no data are changed), *functional* (transient information, i.e., intermediate results), *declarative* (scoped information, i.e., bindings), *imperative* (stable information, i.e., the store), *communicative* (permanent information, i.e., messages send between actions), *directive* (finite representation of self-referential bindings). Actions which process information in more than one facet are called *hybrid.* An action may *commit* and discard alternatives, e.g., in an action A_1 or A_2 representing the nondeterministic choice between two sub-actions, A_1 may commit and thus A_2 is discarded. Compound actions can be build from primitive actions using a special kind of actions called *action combinators.*

Since action semantics provides so many actions and yielders we refrain from giving an exhaustive listing but instead look at some examples.

Similar to denotational semantics the action semantics of a programming language is given by semantic equations[1]:

```
• execute⟦ X ":=" E ⟧ = |evaluate E
                        then
                        |store the given value in the cell bound to X
```

evaluate is a semantic function defined by semantics equations similar to the way the semantic function execute is defined here. For a concrete value of E the function evaluate yields a compound action. The action combinator A_1 then A_2 propagates the transients given to the whole action to A_1, the transients given by A_1 are propagated to A_2, and only the transients given by A_2 are given by the whole action. Thus then represents the left-to-right sequencing in the functional facet. The primitive, imperative action store Y_1 in Y_2 stores the datum produced by the yielder Y_1 in the store cell (a special kind of data) produced by the yielder Y_2. Note, that items of data are a special case of yielders, and always yield themselves when evaluated. In the above example the variable name associated with X in a concrete program would be such a special yielder.

```
• execute⟦ "while" E "do" C "od" ⟧ = unfolding
                                     ||evaluate E
                                     |then
                                     |||execute C then unfold
                                     ||else
                                     |||complete
```

The action combinator unfolding A performs the action A, but whenever it reaches the dummy action unfold it performs A instead. The action complete simply completes and is thus a neutral action with respect to some action combinators. The action combinator A_1 else A_2 is actually syntactic sugar for a compound action:

```
||check the given truth-value and then $A_1$
|or
||check not the given truth-value and then $A_2$
```

The action check Y completes if the yielder Y evaluates to true and fails if it evaluates to false. The yielder not Y evaluates to true (false) if Y evaluates to false (true). The yielder the given D evaluates to a transient datum of sort D given by a preceding action. There can be more than one transient datum, which is taken care of by a labeling mechanism. The action which gives a datum

[1] Instead of using parentheses to indicate precedence of actions, in action semantics we use the convention that vertical lines group actions and their arguments.

can label it, e.g., give Y label $\#n$ and later a yielder can access it, e.g., the given D label $\#n$. Now give Y is short for give Y label $\#0$ and the given D or just the D is short for the given D label $\#0$.

3 The 2BIG Generator

The 2BIG generator [2, 4] applies a sequence of established techniques to a Natural Semantics specification in order to split it into a compiler and an abstract machine. We believe that our framework, by virtue of being compositional, can be extended over time to include even more powerful analysis and transformation methods. Actually, the transformations are mostly source-to-source and after every transformation we have an executable specification again. Of these transformations pass separation is the most important one. Let p be a program and x and y the static and dynamic input to this program, then partial evaluation of p with respect to x yields a residual program p_x, such that $p_x(y) = p(x, y)$. In contrast pass separation transforms the program p into two programs p_1 and p_2 such that $p_2(p_1(x), y) = p(x, y)$. Note that here p_1 produces some intermediate data, which are input to p_2. When it comes to the generation of compiler/executor pairs, pass separation provides an immediate solution, we pass separate the interpreter *interp* into an executor *exec* and a compiler *comp*, such that: $\mathit{interp}(\mathit{prog}, \mathit{data}) = \mathit{exec}(\mathit{comp}(\mathit{prog}), \mathit{data})$. Despite this potential for compiler generation there is only little work on pass separation [7, 6, 3].

Our generator first transforms the 2BIG rules into a term rewriting system:

For this, it first removes side conditions by converting them into transitions, thus there are now only transitions as preconditions. Then it factorizes rules which have a common initial sequence of preconditions. Factorization replaces these rules with a single rule which has the common initial sequence as its preconditions and for each original rule a rule is generated with its remaining preconditions. Next the generator adds a stack to the state in the transitions and stores temporary variables, i.e. variables which are not used in an intermediate transition. Variables which do not occur in the conclusion of a rule are eliminated. The last step before the actual transformation into a term rewriting system is called sequentialization. It converts all preconditions of a rule such that the result state of one transition is the start state of the next. These rules can now be easily turned into rewrite rules. Rules of the form $\frac{c_1 \vdash e_1 \rightarrow e'_1 \quad \ldots \quad c_n \vdash e_n \rightarrow e'_n}{c \vdash e \rightarrow e'}$ are converted into $\langle (c; p), e\rangle \rightarrow \langle (c_1; \ldots; c_n; p), e_1\rangle$ where p is a new variable name.

Now the resulting term rewriting system is in a form, such that pass separation can be applied which yields two term rewriting systems: one representing a compiler and one representing the abstract machine. These term rewriting systems are then further optimized to reduce the number and complexity of the abstract machine instructions, e.g. the number of arguments.

4 Transforming a 2BIG specification of Action Notation

In his PhD thesis [10] deMoura gives a natural semantics specification of a subset of action notation used in the compiler generator Actress [9]. In this specification the order of rules is important. We converted these rules into 2BIG rules adding additional preconditions, when necessary, to make the rules determinate. Then we used our system to generate a compiler and abstract machine represented as term rewriting systems.

In Section 4.1 we demonstrate the generation process by transforming the 2BIG rules for of the GIVE action.

Our specification consists of 100 2BIG rules defining the semantics of 39 action notation constructs including the control, functional, declarative and imperative facets but, as in other Action Semantics directed compiler generators, neither the communicative facet, nondeterminism nor the interleaving of actions. After transformation of side conditions we got 135 rules. Factorization resulted in 191 rules. After sequentialization we got 276 rules. Finally pass separation yielded 216 compiler rules and 276 abstract machine rules. We tested this compiler and abstract machine by translating Mini-Δ programs (e.g., Fibonacci numbers) based on an action semantics specification of the language Mini-Δ [9] into action terms. Then we compiled these action terms using the generated compiler into an abstract machine program and executed the latter by the above abstract machine rules. In other words we use a 2BIG semantics-based compiler generator to generate a compiler and abstract machine for action notation. The generated compiler is then inserted as the back end into a compiler generator based on action semantics (see Figure 1). The front end of this compiler generator was previously developed and used with a positive supercompiler for Prolog[2] as its back end [1].

4.1 Transforming the GIVE Action

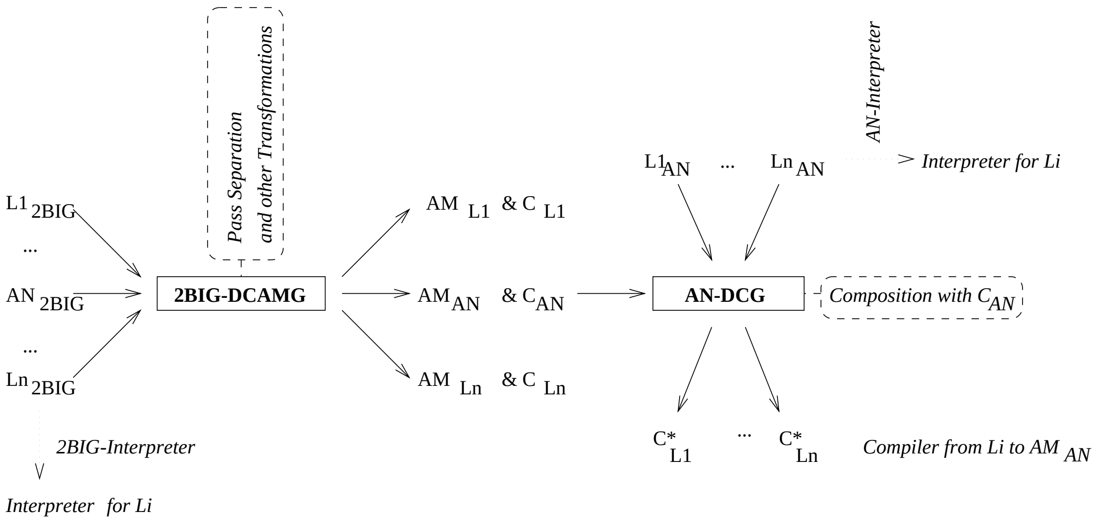
As an example we will now demonstrate step by step how our system generated the abstract machine instructions for the GIVE action.

In the 2BIG specification the following rules define the action *give* which evaluates the yielder Y and returns the resulting value D as a transient. In the rules, states are composed of the transients T, the bindings B and a single-threaded store S. Furthermore there is the outcome status O, which can be *failed* or *completed.*

$$\frac{Y \vdash [T,B,S] \rightarrow datum(D) \qquad D \neq nothing}{give(Y,N) \vdash [T,B,S] \rightarrow [completed,[N \mapsto datum(D)],[],S]}$$

$$\frac{Y \vdash [T,B,S] \rightarrow datum(D) \qquad not(D \neq nothing)}{give(Y,N) \vdash [T,B,S] \rightarrow [failed,[],[],S]}$$

[2] Positive supercompilation [12, 11] is a program specialization technique developed in the functional community. Its adaption to Prolog is not much different from partial evaluation of Prolog [5].



Li_{2BIG}	2BIG specification of language Li
AN_{2BIG}	2BIG specification of action notation
$2BIG - DCAMG$	2BIG directed generator of compilers and abstract machines
AM_{Li}	abstract machine for language Li
AM_{AN}	abstract machine for action notation
C_{Li}	compiler from language Li into AM_{Li}
C_{Li}	compiler from action notation into AM_{Li}
Li_{AN}	action semantics specification of language Li
C^*_{Li}	compiler from language Li into AM_{AN}
$AN - DCG$	action semantics directed compiler generator

Fig. 1. Action-Semantics Directed Compiler Generation

There are two side conditions in the above rules, one is the negation of the other. Transforming the side conditions yields:

$$\frac{Y \vdash [T,B,S] \to datum(D) \qquad test_1 \vdash [D] \to true}{give(Y,N) \vdash [T,B,S] \to [completed,[N \mapsto datum(D)],[],S]}$$

$$\frac{Y \vdash [T,B,S] \to datum(D) \qquad test_1 \vdash [D] \to false}{give(Y,N) \vdash [T,B,S] \to [failed,[],[],S]}$$

$test_1 \vdash [D] \to D \neq nothing$

After factorization of the above rules we have:

$$\frac{Y \vdash [T,B,S] \to datum(D) \qquad test_1 \vdash [D] \to R \qquad fact_{give}(N) \vdash [[D,S],R] \to E}{give(Y,N) \vdash [T,B,S] \to E}$$

$fact_{give}(N) \vdash [[D,S],true] \to [completed,[N \mapsto datum(D)],[],S]$

$fact_{give}(N) \vdash [[D,S],false] \to [failed,[],[],S]$

$test_1 \vdash [D] \to D \neq nothing$

Now the stack (Z) is introduced and temporary variables are allocated:

$$\frac{\begin{array}{l} Y \vdash [[S|Z],[T,B,S]] \to [[S|Z],datum(D)] \\ test_1 \vdash [[[S,D]|Z],[D]] \to [[[S,D]|Z],R] \quad fact_{give}(N) \vdash [Z,[[D,S],R]] \to [Z,E] \end{array}}{give(Y,N) \vdash [Z,[T,B,S]] \to [Z,E]}$$

$fact_{give}(N) \vdash [Z,[[D,S],true]] \to [Z,[completed,[N \mapsto datum(D)],[],S]]$

$fact_{give}(N) \vdash [Z,[[D,S],false]] \to [Z,[failed,[],[],S]]$

$test_1 \vdash [Z,[D]] \to [Z,D \neq nothing]$

Next these rules can be sequentialized:

$$\frac{\begin{array}{ll} Y \vdash [[S|Z],[T,B,S]] \to [[S|Z],datum(D)] & \\ conv_5 \vdash [[S|Z],datum(D)] \to [[[S,D]|Z],[D]] & test_1 \vdash [[[S,D]|Z],[D]] \to [[[S,D]|Z],R] \\ conv_6 \vdash [[[S,D]|Z],R] \to [Z,[[D,S],R]] & fact_{give}(N) \vdash [Z,[[D,S],R]] \to [Z,E] \end{array}}{give(Y,N) \vdash [Z,[T,B,S]] \to [Z,E]}$$

$fact_{give}(N) \vdash [Z,[[D,S],true]] \to [Z,[completed,[N \mapsto datum(D)],[],S]]$

$fact_{give}(N) \vdash [Z,[[D,S],false]] \to [Z,[failed,[],[],S]]$

$test_1 \vdash [Z,[D]] \to [Z,D \neq nothing]$

$conv_2 \vdash [[S|Z],datum(D)] \to [[[S,D]|Z],[D]]$

$conv_3 \vdash [[[S,D]|Z],R] \to [Z,[[D,S],R]]$

Now a term rewriting system is generated:

$$\begin{array}{ll}
\langle give(Y,N); C, [Z,[T,B,S]]\rangle & \\
\qquad \Longrightarrow \ \langle Y; conv_2; test_1; conv_3; fact_{give}(N); C, [[[S]|Z],[T,B,S]]\rangle & \\
\langle fact_{give}(N); C, [Z,[[D,S], true]] & \Longrightarrow \langle C, [Z, [completed, [N \mapsto datum(D)], [], S]]\rangle \\
\langle fact_{give}(N); C, [Z,[[D,S], false]] & \Longrightarrow \langle C, [Z, [failed, [], [], S]]\rangle \\
\langle test_1; C, [Z,[D]] & \Longrightarrow \langle C, [Z, D \neq nothing]\rangle \\
\langle conv_2; C, [[[S]|Z], datum(D)]\rangle & \Longrightarrow \langle C, [[[S,D]|Z],[D]]\rangle \\
\langle conv_3; C, [[[S,D]|Z], R]\rangle & \Longrightarrow \langle C, [Z,[[D,S],R]]\rangle
\end{array}$$

Finally we apply the pass separation transformation and we get the following compiler rules:

$$\begin{array}{ll}
give(Y,N) & \Longrightarrow \overline{give}(Y,N); Y; conv_2; test_1; conv_3; fact_{give}(N) \\
fact_{give}(N) & \Longrightarrow \overline{fact}_{give}(N) \\
test_1 & \Longrightarrow \overline{test}_1 \\
conv_2 & \Longrightarrow \overline{conv}_2 \\
conv_3 & \Longrightarrow \overline{conv}_3
\end{array}$$

And the following abstract machine rules:

$$\begin{array}{ll}
\langle \overline{give}(Y,N); C, [Z,[T,B,S]]\rangle & \Longrightarrow \langle C, [[[S]|Z],[T,B,S]]\rangle \\
\langle \overline{fact}_{give}(N); C, [Z,[[D,S], true]]\rangle & \Longrightarrow \langle C, [Z, [completed, [N \mapsto datum(D)], [], S]]\rangle \\
\langle \overline{fact}_{give}(N); C, [Z,[[D,S], false]]\rangle & \Longrightarrow \langle C, [Z, [failed, [], [], S]]\rangle \\
\langle \overline{test}_1; C, [Z,[D]]\rangle & \Longrightarrow \langle C, [Z, D \neq nothing]\rangle \\
\langle \overline{conv}_2; C, [[[S]|Z], datum(D)]\rangle & \Longrightarrow \langle C, [[[S,D]|Z],[D]]\rangle \\
\langle \overline{conv}_3; C, [[[S,D]|Z], R]\rangle & \Longrightarrow \langle C, [Z,[[D,S],R]]\rangle
\end{array}$$

5 Prototyping Tools

In Figure 1 we show how the different generators and interpreters can be used for both rapid prototyping of language specifications and generation of compilers and abstract machines. First we can use a 2BIG-interpreter to test a 2BIG-specification Li_{2BIG} of programming languages Li. Then we can generate an abstract machine AM_{Li} for the language Li and a compiler C_{Li} from Li to AM_{Li} using our 2BIG-semantics directed compiler and abstract machine generator (2BIG-DCAMG). The generator's central transformation is pass separation of term rewriting rules. In addition it applies many pre- and post-processing transformations including several optimizations.

Based on a 2BIG-specification AN_{2BIG} of a certain language, namely action notation, we generate a compiler and abstract machine for action notation. Now an action semantics specification Li_{AN} of a programming language Li can be tested both by using an action notation interpreter or by composing the action notation specification,i.e., semantics equations mapping Li programs to action notation terms, with the compiler C_{AN}. This composition results in an action semantics-directed compiler generator (AN-DCG).

Our prototyping environment includes several tools written in Prolog:

– a 2BIG interpreter
– an action notation interpreter (actually we have a handwritten interpreter, but we can also use the 2BIG interpreter to execute action terms using the 2BIG specification of action notation)
– an interpreter for compiler and abstract machine rules
– a compiler of source language programs to C using the compiler and abstract machine rules
– a compiler of compiler rules and abstract machine rules to SML

6 An Abstract Machine Language Language

Since Action Semantics is a formal language to define programming languages, we expect, that the abstract machine language AM_{AN} generated for Action Semantics is suitable to define abstract machine languages. Rather than just composing the AM_{AN} and the semantics equation which gives us AN-DCG, we could try a method similar to the combinator based approach of Wand [13, 14]. Given an Action Semantics specification of a programming language L:

1. Translate the right hand sides of the semantics equation using the generated compiler into AM_{AN} (this results in an AN-DCG).
2. Look for recurring patterns in the translated right hand side.
3. Define new instructions based on these patterns. These new instructions form an abstract machine specific for L.
4. Fold the patterns in the semantics equations by the new instructions. The resulting equations constitute a compiler into the abstract machine for L.

7 Action Semantics-Directed Compiler Generation

Now we will show how our action semantics-based compiler generator works by means of a simple example. The semantics of the language Mini-Δ is given by equations like the following one:

- execute⟦ X ":=" E ⟧ =
  |evaluate E
  then store the value in the cell bound to X

These semantic equations define a translation function from source language programs to action terms. Using this action semantics specification of Mini-Δ the following program

```
let const i=1;
    var   x:integer;
in x:=2+i end
```

is translated into the following action term

```
• |furthermore
  |||give num(1) then bind i to the given value
  ||before
  |||allocate a cell of type integer then bind x to the cell
  hence
  ||||give num(2) then give the value label#1
  |||and
  ||||||give the value stored in the cell bound to i
  |||||or
  ||||||give the value bound to i
  ||||then
  |||||give the value label #2
  ||then
  |||give add(the value #1,the value #2)
  |then
  ||store the given value in the cell bound to x
```

In our system we use prefix notation instead of the mixfix notation usually used for action terms. Thus A_1 **then** A_2 becomes **then**(A_1, A_2). The above action term in prefix notation is:

```
hence(
  furthermore(
      before(then(give(num(1),0),bind(i,the(value,0))),
             then(allocate(cell(integer)),bind(x,the(cell,0))))),
   then(
     then(
       and(then(give(num(2),0),give(the(value,0),1)),
           then(or(give(stored(value,bound(cell,i)),0),
                   give(bound(value,i),0)),
                give(the(value,0),2))),
       give(add(the(value,1),the(value,2)),0)),
     store(the(value,0),bound(cell,x))))
```

Now this action term is converted into a very long abstract machine program by the generated compiler. One reason for the length of the abstract machine program is that recurring subprograms are not shared.

$$
\begin{array}{l}
\overline{hence}(\\
\quad \overline{furthermore}(\\
\quad\quad \overline{before}(\\
\quad\quad\quad \overline{then}(\\
\quad\quad\quad\quad (\overline{give}(\overline{num}(1), 0); \\
\quad\quad\quad\quad \overline{num}(1); \\
\quad\quad\quad\quad \overline{conv}_2; \\
\quad\quad\quad\quad \overline{test}_1; \\
\quad\quad\quad\quad \overline{conv}_3; \\
\quad\quad\quad\quad \overline{fact}_{give}(0)), \\
\quad\quad\quad\quad (\overline{bind}(i, \overline{the}(value, 0)); \\
\quad\quad\quad\quad \overline{the}(value, 0); \\
\quad\quad\quad\quad \overline{conv}_1 4; \\
\quad\quad\quad\quad \overline{test}_5; \\
\quad\quad\quad\quad \overline{conv}_1 5; \\
\quad\quad\quad\quad \overline{fact}_{bind}(i))); \\
\quad\quad\quad\quad \overline{give}(\overline{num}(1), 0); \\
\quad\quad\quad\quad \ldots
\end{array}
$$

The execution of the above program by the abstract machine in the empty environment yields the expected result: a memory cell is allocated for the variable x and the value 3 is stored in it.

In the above example the action term could be simplified before translating it into the abstract machine language. As an example

give num(2) then give the value label#1

can be simplified to give num(2) label#1. Analyses and simplifications of action terms have been investigated in de Moura's PhD thesis [10]. It would be interesting to use the simplified action terms produced by his Actress system and translate those into the generated abstract machine language.

8 Experimental Results for Optimizations

For the action notation specification, the optimizations of the generated term rewriting systems lead to a significant reduction of the number of rules both of the compiler and the abstract machine. First, by self-application, the number of compiler rules was reduced from 216 to 43. Second, using the other optimizations we got 181 instead of 276 abstract machine rules.

$$give(Y, N) \Longrightarrow \overline{give}(Y, N); Y; conv_2; test_1; conv_3; fact_{give}(N)$$

$$give(Y, N) \Longrightarrow \overline{give}; Y; \overline{comb}; \overline{fact}_{disp}(factor_{give}, N)$$

Comparing the original and the optimized compiler rule for the GIVE action we find that the following optimizations have been applied:

– The arguments to the abstract machine instruction $\overline{give}$ have been removed.
– There are no more compiler rules for $conv_2$, $test_1$, etc.
– The sequence of instructions $\overline{conv}_2;\overline{test}_1;\overline{conv}_3$ has been combined into the instruction $\overline{comb}$.
– Some abstract machine rules of $factor_{give}$ have been conflicting with rules of other instructions and thus these term rewriting rules have been factorized. This lead to the introduction of the new instruction $\overline{fact}_{disp}$.

9 Conclusion

So far our prototyping tools have been used to implement a considerable subset of Action Semantics. Instead one could also try to implement subsets of Action Semantics restricted to a few facets. As an example, to specify functional languages we could implement a version of Action Semantics without the imperative facet. As a consequence the generated abstract machine would not have a store as a compenent of its state. Another approach would be to implement an annotated version of Action Semantics and a preprocessing phase, e.g., a binding-time analysis, which translates action terms into annotated terms. Finally, rather than just experimenting with existing semantics formalism, our system can also be used to design and implement new semantics formalisms.

References

[1] Stephan Diehl. A Prolog Positive Supercompiler. in Proceedings of "Arbeitstagung Programmiersprachen", Herbert Kuchen, editor, published as Arbeitsbericht No. 58 of the Institut für Wirtschaftsinformatik, Westfälische Wilhelms-Universität Münster, 1997

[2] Stephan Diehl. *Semantics-Directed Generation of Compilers and Abstract Machines.* PhD thesis, University Saarbrücken, Germany, 1996. http://www.cs.uni-sb.de/~diehl/phd.html.

[3] Stephan Diehl. Transformations of Evolving Algebras. In *Proceedings of VIII International Conference on Logic and Computer Science LIRA'97*, pages 43–50, Novi Sad,Yugoslavia, 1997.

[4] Stephan Diehl. Natural Semantics Directed Generation of Compilers and Abstract Machines. *Formal Aspects of Computing (to appear)*, 1999.

[5] R. Glück and M.H. Sørensen. Partial Deduction and Driving are Equivalent. In *PLILP'94*. 1994.

[6] John Hannan. Operational Semantics-Directed Compilers and Machine Architectures. *ACM Transactions on Programming Languages and Systems*, 16(4):1215–1247, 1994.

[7] U. Jørring and W.L. Scherlis. Compilers and Staging Transformations. In *13th ACM Symposium on Principles of Programming Languages*, 1986.

[8] P.D. Mosses. *Action Semantics.* Cambridge University Press, 1992.

[9] H. Moura and D. A. Watt. Action Transformations in the ACTRESS Compiler Generator. In *CC'94*, volume LNCS 768. Springer Verlag, 1994.

[10] Hermano Moura. *Action Notation Transformations.* PhD thesis, University of Glasgow, 1993.

[11] M.H. Sørensen, R. Glück, and N. D. Jones. Towards Unifying Partial Evaluation, Deforestation, Supercompilation and GPC. In D. Sannella, editor, *Programming Languages and Systems*, volume LNCS 788. Springer Verlag, 1994.

[12] V.F. Turchin. The Concept of a Supercompiler. *ACM TOPLAS*, 8(3), 1986.

[13] Mitchell Wand. Semantics-Directed Machine Architecture. In *Proc. of POPL'82.* 1982.

[14] Mitchell Wand. From Interpreter to Compiler: A Representational Derivation. In H. Ganzinger, N.D. Jones, editor, *Programs as Data Objects*, volume LNCS 217. Springer Verlag, 1986.

Online Partial Evaluation of Actions*

Kyung-Goo Doh[1] and Hyun-Goo Kang[2]

[1] Hanyang University, Korea
doh@cse.hanyang.ac.kr
http://pllab.hanyang.ac.kr/∼doh

[2] Electronics and Telecommunications Research Institute, Korea
hgkang@etri.re.kr
http://pllab.hanyang.ac.kr/∼hgkang

Abstract. In compiler-generation systems based on action semantics, it is important to statically process the expanded action denotation – used as an intermediate code – as much as possible so that the generated compiler can produce better object code. In fact, it is shown in the Brown-Moura-Watt's ACTRESS system that the action transformation by eliminating transient and binding actions greatly improve the efficiency of object code.
In this paper, we present an automatic action-transformation method based on online partial evaluation. The previous off-line method cannot partially evaluate actions inside the body of unfolding-action and abstraction without performing separate global analysis. The proposed online method remedies the problem, thus naturally improves the quality of residual action. We also extend the method so that imperative actions can be partially evaluated.

1 Introduction

Action semantics is a framework for formally specifying programming languages [6, 13]. Action-semantics-directed compiler generators take an action semantics definition of a programming language and automatically generate a compiler of the language. The generated compiler first expands a source program into an action denotation of the program, and then compiles the action denotation to a target program. An action-semantics-directed compiler generation system, Actress [1], employs some transformation rules along with algebraic simplification to eliminate actions that give statically known values as well as actions that produce statically removable bindings. Doh [2] devised an automatic transformation method based on off-line partial evaluation [10] using two-level type system. However, his off-line method cannot partially evaluate actions inside the body of unfolding-action and abstraction without performing separate global analysis.

* This work was supported in part by grant No.971-0903-026-2 from the Basic Research program of the Korea Science and Engineering Foundation and by grant No.96171-IT1-I1 from the University Research Program supported by Ministry of Information & Communication in South Korea.

In this paper we propose an online partial-evaluation method [10] which remedies the problem, thus naturally improves the quality of residual action. We also extend the method so that imperative actions can be partially evaluated.

The reader of this paper is assumed to be somewhat familiar with action semantics. Curious readers who want the full exposure of the framework will find inventors' books and tutorials [6, 7, 13] useful. The rest of the paper is organized as follows. Section 2 describes the subset of action notation used in this paper. Section 3 explains the kind of action transformation to be achieved. Section 4 presents how the online partial evaluation is done. Section 5 shows test results. Section 6 discusses related works. Section 7 concludes.

2 Action notation

Action notation was developed by Mosses and Watt as a semantic meta-language to give meanings to programs [6, 13]. Each action represents the implementation-independent computational behavior of programs. Actions operate upon different kinds of *facets*: functional-facet actions manipulate data (transient information), declarative-facet actions create and access the bindings of tokens to data (scoped information), and imperative-facet actions allocate and manipulate primary storage (stable information). Action notation consists of primitive actions (representing a single computation step), action combinators (defining the data flow and control flow of actions), and yielders (evaluated to yield data during action performance).

```
Yielder =

    Integer | Truthvalue | cell(natural) | binop(Yielder,Yielder) |
    it | them | it#natural | the data bound to Token |
    closure abstraction of Action | the integer stored in Yielder |
    (Yielder,Yielder,...,Yielder)

Action =

    give Yielder | enact application Yielder to Yielder |
    bind Token to Yielder |
    reserve a cell | allocate a cell | store Yielder in Yielder |
    Action and then Action | Action then Action |
    furthermore Action hence Action | Action before Action |
    Yielder then either Action or Action |
    unfolding Action | unfold | complete
```

Fig. 1. Syntax of action notation

The syntax of action notation used in this paper is shown in Figure 1. [1] Constant yielders, e.g., `true`, `3`, `cell(7)`, yield themselves. Yielders, `it` and `them`, yield data given to them. The only difference between the two is that the former is only allowed to deal with a single datum. The yielder `it`$\#n$ yields the nth component of the given tuple. `the data bound to` I yields the data bound to the token I in the given bindings. `the integer stored in` Y yields the integer value stored in the cell yielded by Y. Notice that the integer value is the only storable data in our notation. The compound yielders, such as `sum(3,the data bound to "x")`, etc., yield the result of applying the operation to the data yielded by its operands. `closure abstraction of` A gives an abstraction that incorporates A with the current bindings, which forces static scoping. Dynamic scoping does not always guarantee safe transformation. Thus, the action notation in this paper is only limited to static scoping, meaning that all the bindings we deal with in this paper are statically scoped. Some criteria to determine whether a particular action is statically scoped or dynamically scoped is described in detail in [8, 9]. `give` Y gives the data yielded by Y. `enact application` Y_1 `to` Y_2 performs the action incorporated in the abstraction yielded by Y_1. The data yielded by Y_2 and the bindings at the time of incorporating the action are available to the performing action. `bind` I `to` Y produces the binding of a token I to the data yielded by Y. `store` Y_1 `in` Y_2 stores the data yielded by Y_1 in the cell yielded by Y_2. `allocate a cell` allocates a new cell and gives the cell. `unfolding` A performs A; and when `unfold` is encountered, it performs A again. Y `then either` A_1 `or` A_2 performs A_1 if Y yields `true`; A_2 performs if Y yields `false`. A_1 `and then` A_2 performs sequentially and combines the results of both actions. A_1 `then` A_2 also performs sequentially, but the data given by A_1 is given to A_2 and the result of A_2 is that of the whole action. `furthermore` A_1 `hence` A_2 represents an ordinary block structure, in which the received bindings are overridden by the ones produced by A_1 and then given to A_2. A_1 `before` A_2 represents the sequencing of declarations. The received bindings are overridden by the ones produced by A_1 and given to A_2. Then, the whole action produces the bindings produced by A_1 overridden by the ones by A_2.

The semantics of action notation has been described informally in English. As the reader might have noticed, most of the notations more-or-less explain themselves.

3 Action transformation

Given the action semantics definition of a programming language, a naive compiler-generation system generates a program (called an *expander*) that expands a source program into an action program (called *action denotation*) representing the meaning of the source program. The expanded action denotation is normally

[1] The action notation used in this paper is by no means complete. The subset has been carefully chosen just enough to experiment with automatic action transformation by partial evaluation. For a formal, comprehensive description of the notation, the reader should refer to Mosses' book, "Action Semantics" [6].

viewed as an intermediate representation in compiler-generation systems. The action denotation is normally huge in size compared to the original source program because it is mainly designed to give meanings to the program. Thus the naively expanded action program naturally contains actions that can be performed at compile-time. Even though some previous experiments (ACTRESS [1], Cantor[12], Oasis[11], and Genesis [4]) showed that systems employing an action semantics approach are still favorable to the counterparts based on traditional denotation semantics, there still is room for improvement.

Hence, the goal of this paper is to do a meaning-preserving transformation of action by processing as many actions involving static computation as possible. Consider an action, `give 3 then give sum(it,2)`. The known datum 3 given by the left subaction `give 3` is passed to the right subaction `give sum(it,2)`. It means that the datum 3 is consumed by the right subaction and is not propagated beyond the action (i.e., its life is over as soon as it is consumed by the right subaction). Thus the action can be transformed into an action with the same meaning, `give sum(3,2)`, which, in turn, is safely transformed into `give 5` by evaluating `sum(3,2)`. This kind of transformation is called *transient elimination* in [8, 9]. Consider another example.

```
| give 4 and then give the data bound to "x"
then give sum(it#1,it#2)
```

If the data bound to token `"x"` is known, say 3, then the above action can surely be transformed into `give 7`. On the other hand, if the data bound to `"x"` is unknown, only the known datum 4 is propagated, eliminating `give 4` safely as follows:

```
| give the data bound to "x"
then give sum(4,it#1)
```

Notice that the index `#2` is changed to `#1`. Note also that, in this particular example, it seems to be a good idea to transform the above action even further into:

```
give sum (4,the data bound to "x")
```

However, this kind of transformation does not always do good. For example, if an action

```
| give the data bound to "x"
then give sum(it,sum(it,it))
```

might be transformed into

```
give sum(the data bound to "x",
     sum(the data bound to "x",
         the data bound to "x"))
```

In this case, the number of access to the bindings has been increased. Thus, this paper does not employ this kind of transformation, but concentrates only with consuming and propagating static data.

Next, we consider some binding actions to eliminate (called *binding elimination* in [8, 9]). (Recall that bindings are forced to be statically scoped in this paper.) For example, `bind "x" to 7 hence give the data bound to "x"` can be safely transformed to `give 7` which completely eliminates the use of token `"x"` at run-time. For another, yet more challenging example of binding elimination, consider:

```
| furthermore
| | | bind "p" to closure abstraction of give sum(it#1,it#2)
| | before
| | | bind "x" to 3
hence
| ...
| enact application the abstraction bound to "p"
|       to (the data bound to "x",the data bound to "y")
| ...
```

which should be safely transformed to:

```
| furthermore
| | bind "p3" to closure abstraction of give sum(3,it#1)
hence
| ...
| enact application the abstraction bound to "p3"
|       to the data bound to "y"
| ...
```

In addition to the transient and binding elimination discussed so far, imperative actions are also partially evaluated, which will be discussed in the next section.

The rest of this paper discusses how the action transformations discussed in this section can be done automatically by online partial evaluation.

4 Online partial evaluation

We now explain how to transform actions using online partial-evaluation technique. At partial-evaluation time, actions are performed, if possible, using available data, bindings and/or storage. Otherwise, actions remain as a code called *residual action*. For example, the `give 3` action may be completely performed giving a static value 3. Meanwhile, the `give it` action may be either completely performed or not performed at all depending on the given value. If the given value is known, it can be performed and giving the value. Otherwise, it remains as a code. In this section, we discuss the online partial evaluation technique for actions.

4.1 Semantic domain

In action notation, transients are the tuples of data. The data may be either an integer, a truth value, a cell, or an abstraction. Since the data processed by partial evaluation can be the given data and/or the code(residual yielder), it is necessary to distinguish them from each other using an appropriate mark. In this paper, the known (static) data v is represented as $\langle \mathrm{S}, v\rangle$, and the code(residual yielder) as $\langle \mathrm{D}, y, i\rangle$, where i is the index in the tuple the yielder belongs to. An abstraction is a closure containing an action, bindings and a marker. The marker indicates whether or not the abstraction is bound to an identifier. This information is used to partially evaluate `enact ...` actions. The semantic domain for transients and bindings is as follows:

$$\text{Data} = \text{Integer} \cup \text{Truth-value} \cup \text{Cell} \cup \text{PE-Abstraction} \cup \{\text{unknown}\}$$

$$\text{PE-Abstraction} = \text{Action} \times \text{PE-Bindings} \times \text{Is-Bound}$$

$$\text{PE-Data} = (\{\mathrm{S}\} \times \text{Data}) \cup (\{\mathrm{D}\} \times \text{Yielder} \times \text{Index})$$

$$\text{PE-Transients} = \text{PE-Data} \times \cdots \times \text{PE-Data}$$

$$\text{PE-Bindings} = (\text{ID} \mapsto \text{PE-Data}) \times \cdots \times (\text{ID} \mapsto \text{PE-Data})$$

The semantic domain for storage is as follows:

$$\text{PE-Storage} = \text{Location} \times \text{Storable} \times \text{SD-Marker}$$

$$\text{Storable} = \text{Integer} \cup \{\bot\}$$

Only integer values are storable, and each cell is marked with S/D and SS/DD showing the status of (the history) of the value stored in the cell. Each marker has the meaning as follows:

- S: the current value stored in the cell is static.
- D: the current value stored in the cell is dynamic.
- SS: the values that have been stored in the cell so far are all static.
- DD: some values that have been stored in the cell so far are dynamic.

This information is used to residualize imperative actions.

4.2 Yielders

The judgement giving the partial evaluation of yielders has the form:

$$(t, b, s, d) \vdash Y \Rightarrow t'$$

where $Y \in$ Yielder, $t, t' \in$ PE-Data, $b \in$ PE-Bindings, $s \in$ Storage and $d \in$ Depth.

The partial evaluation rules for yielders are shown in Fig. 2. Primitive data, `n` ($\in$ Integer), `true`, `false` and `cell(l)` are always known, and so the results are $\langle \mathrm{S}, n\rangle$, $\langle \mathrm{S}, \mathit{true}\rangle$, $\langle \mathrm{S}, \mathit{false}\rangle$, and $\langle \mathrm{S}, \mathit{cell}(l)\rangle$, respectively.

$$\_ \vdash n \Rightarrow \langle \mathrm{S}, n\rangle \qquad \_ \vdash \texttt{true} \Rightarrow \langle \mathrm{S}, \mathit{true}\rangle$$

$$\_ \vdash \texttt{false} \Rightarrow \langle \mathrm{S}, \mathit{false}\rangle \qquad \_ \vdash \texttt{cell(l)} \Rightarrow \langle \mathrm{S}, \mathit{cell}(l)\rangle$$

$$\frac{(t,b,s,d) \vdash Y_1 \Rightarrow \langle \mathrm{S}, v_1\rangle \qquad (t,b,s,d) \vdash Y_2 \Rightarrow \langle \mathrm{S}, v_2\rangle}{(t,b,s,d) \vdash \mathit{binop}(Y_1,Y_2) \Rightarrow \langle \mathrm{S}, [\![\mathit{binop}]\!](v_1,v_2)\rangle}$$

$$\frac{(t,b,s,d) \vdash Y_1 \Rightarrow \langle \mathrm{D}, y_1, i_1\rangle \qquad (t,b,s,d) \vdash Y_2 \Rightarrow \langle \mathrm{S}, v_2\rangle}{(t,b,s,d) \vdash \mathit{binop}(Y_1,Y_2) \Rightarrow \langle \mathrm{D}, \mathit{binop}(y_1, \langle \mathrm{S}, v_2\rangle \Uparrow), 1\rangle}$$

$$\frac{(t,b,s,d) \vdash Y_1 \Rightarrow \langle \mathrm{S}, v_1\rangle \qquad (t,b,s,d) \vdash Y_2 \Rightarrow \langle \mathrm{D}, y_2, i_2\rangle}{(t,b,s,d) \vdash \mathit{binop}(Y_1,Y_2) \Rightarrow \langle \mathrm{D}, \mathit{binop}(\langle S, v_1\rangle \Uparrow, y_2), 1\rangle}$$

$$\frac{(t,b,s,d) \vdash Y_1 \Rightarrow \langle \mathrm{D}, y_1, i_1\rangle \qquad (t,b,s,d) \vdash Y_2 \Rightarrow \langle \mathrm{D}, y_2, i_2\rangle}{(t,b,s,d) \vdash \mathit{binop}(Y_1,Y_2) \Rightarrow \langle \mathrm{D}, \mathit{binop}(y_1, y_2), 1\rangle}$$

$$\begin{array}{rcl}
(t,b,s,d) \vdash \texttt{it} & \Rightarrow & \text{case } t \text{ of} \\
& & \quad \langle \mathrm{S}, v\rangle \rightarrow \langle \mathrm{S}, v\rangle \\
& & \mid \langle \mathrm{D}, y, i\rangle \rightarrow \langle \mathrm{D}, \texttt{it}, 1\rangle \\
(t,b,s,d) \vdash \texttt{it\#}n & \Rightarrow & \text{case } \mathit{component\#}(n,t) \text{ of} \\
& & \quad \langle \mathrm{S}, v\rangle \rightarrow \langle \mathrm{S}, v\rangle \\
& & \mid \langle \mathrm{D}, y, i\rangle \rightarrow \langle \mathrm{D}, \texttt{it\#}i, 1\rangle \\
(t,b,s,d) \vdash \texttt{them} & \Rightarrow & t
\end{array}$$

$$\begin{array}{rcl}
(t,b,s,d) \vdash \texttt{the data bound to } I & \Rightarrow & \text{case } \mathit{blookup}(I,b) \text{ of} \\
& & \quad \langle \mathrm{D}, y, i\rangle \rightarrow \langle \mathrm{D}, \texttt{the data bound to } I, 1\rangle \\
& & \mid \langle \mathrm{S}, v\rangle \rightarrow \langle \mathrm{S}, v\rangle
\end{array}$$

$$\frac{(t,b,s,d) \vdash Y \Rightarrow \langle \mathrm{S}, \mathit{cell}(l)\rangle \qquad \mathit{sd}(l,s) = \mathrm{S}}{(t,b,s,d) \vdash \texttt{the integer stored in } Y \Rightarrow \langle \mathrm{S}, \mathit{slookup}(l,s)\rangle}$$

$$\frac{(t,b,s,d) \vdash Y \Rightarrow \langle \mathrm{S}, \mathit{cell}(l)\rangle \qquad \mathit{sd}(l,s) = \mathrm{D}}{\begin{array}{l}(t,b,s,d) \vdash \texttt{the integer stored in } Y \Rightarrow \\ \qquad \langle \mathrm{D}, \texttt{the integer stored in } \langle \mathrm{S}, \mathit{cell}(l)\rangle \Uparrow, 1\rangle\end{array}}$$

$$\frac{(t,b,s,d) \vdash Y \Rightarrow \langle \mathrm{D}, y, i\rangle}{(t,b,s,d) \vdash \texttt{the integer stored in } Y \Rightarrow \langle \mathrm{D}, \texttt{the integer stored in } y, 1\rangle}$$

$$\frac{(t,b,s,d) \vdash y_i \Rightarrow w_i \qquad (\mathit{where}\ i \in \{1..n\})}{(t,b,s) \vdash (y_1,..,y_n) \Rightarrow (w_1,..,w_n)}$$

Fig. 2. Yielders

The first binary-operator rule covers the case where all the argument data are known. In this case, we can simply evaluate both yielders and give the results of applying the operator to the data yielded by both yielders (marked as S). The second, third, and fourth rules deal with the case where at least one of the arguments is unknown (that is, at least one of them is marked D). In this case, the binary operator cannot be evaluated, and thus it must be reconstructed. When being reconstructed, any of the data marked S must be converted to the yielder, which is called *lifting.* The lifting rules are as follows:

$$\langle \mathrm{S}, v\rangle \ \Uparrow [[v]]^{-1} \qquad \text{where } v \in \text{ Data}$$

$$\langle \mathrm{D}, y, i\rangle \ \Uparrow \ y \qquad \text{where } y \in \text{ Yielder}$$

$$\frac{w_i \ \Uparrow \ y_i}{(w_1, .., w_n) \ \Uparrow \ (y_1, .., y_n)} \qquad \text{where } i \in \{1..\text{n}\}$$

If the PE-Data is static $\langle \mathrm{S}, v\rangle$, the yielder corresponding to v is constructed. If the PE-Data is dynamic $\langle \mathrm{D}, y, i\rangle$, then the yielder y is simply returned.

The yielder `it` is simply yields what is given. Thus, the partial evaluation does the same. In case of the yielder `it#`n which gives nth component of the given data tuple, if the nth component is $\langle \mathrm{D}, y, i\rangle$, then the result is $\langle \mathrm{D}, \texttt{it\#}i, 1\rangle$. Notice that the index has changed to i from n, which keeps the correct index when some of the given tuple components are static and removed from the tuple. For example,

```
| give 3 and then give the data bound to "x"
then give it#2
```

partially evaluates to, if the data bound to x is unknown,

```
| give the data bound to "x"
then give it#1
```

The yielder `the data bound to` I yields the data bound to I in the given bindings as it is. The yielder `the integer stored in` Y yields the value stored in the cell yielded by Y. As the third rule shows, if the cell yielded by Y is unknown, the residual yielder is constructed accordingly. If the cell is known, then the result is different depending on the status of the integer value stored in the cell. If the value is known, then the value with a mark S is returned (the first rule). Otherwise, the residual yielder is constructed accordingly (the second rule).

For a tuple, each component is partially evaluated and then tupled.

4.3 Primitive actions

The judgement giving the partial evaluation of actions has the form:

$$(t, b, s, \delta) \vdash A \Rightarrow (t', b', s', a)$$

$$(t,b,s,\delta) \vdash \texttt{complete} \;\Rightarrow\; (\emptyset,\emptyset,s,\texttt{complete})$$

$$\frac{\mathit{eliminable}(\delta) \quad (t,b,s,d) \vdash Y \;\Rightarrow\; w \quad \mathit{all\text{-}static}(w)}{(t,b,s,\delta) \vdash \texttt{give}\ Y \;\Rightarrow\; (w,\emptyset,s,\texttt{complete})}$$

$$\frac{\mathit{eliminable}(\delta) \quad (t,b,s,d) \vdash Y \;\Rightarrow\; w \quad \mathit{not\ all\text{-}static}(w)}{(t,b,s,\delta) \vdash \texttt{give}\ Y \;\Rightarrow\; (w,\emptyset,s,\mathit{residualize}(w,\delta_e))}$$

$$\frac{\mathit{not\ eliminable}(\delta) \quad (t,b,s,d) \vdash Y \;\Rightarrow\; w}{(t,b,s,\delta) \vdash \texttt{give}\ Y \;\Rightarrow\; (w,\emptyset,s,\mathit{residualize}(w,\delta_e))}$$

$$\frac{\mathit{eliminable}(\delta) \quad (t,b,s,d) \vdash Y \;\Rightarrow\; \langle \mathrm{S}, v\rangle}{(t,b,s,\delta) \vdash \texttt{bind}\ I\ \texttt{to}\ Y \;\Rightarrow\; (\emptyset,\{I \mapsto \langle \mathrm{S}, v'\rangle\},s,\texttt{complete})}$$

where $v' = \text{case } v \text{ of}$
$\quad\quad \mathit{abstraction}(a',b',\_) \Rightarrow \mathit{abstraction}(a',b',\mathit{bound})$
$\quad | \ \_ \Rightarrow v$

$$\frac{(t,b,s,d) \vdash Y \;\Rightarrow\; w}{(t,b,s,\delta) \vdash \texttt{bind}\ I\ \texttt{to}\ Y \;\Rightarrow\; (\emptyset,\{I \mapsto w\},s,\texttt{bind}\ I\ \texttt{to}\ w \Uparrow)}$$

$$\frac{\mathit{statically\_allocatable}(\delta) \quad \mathit{allocate}(s) \;=\; (l,s')}{(t,b,s,\delta) \vdash \texttt{allocate cell} \;\Rightarrow\; (\langle \mathrm{S}, \mathit{cell}(l)\rangle,\emptyset,s',\texttt{reserve a cell})}$$

$$\frac{\mathit{not\ statically\_allocatable}(\delta)}{(t,b,s,\delta) \vdash \texttt{allocate cell} \;\Rightarrow\; (\langle \mathrm{D}, \texttt{given cell\#}1,1\rangle,\emptyset,s,\texttt{allocate a cell})}$$

$$\frac{(t,b,s,d) \vdash Y_1 \;\Rightarrow\; \langle \mathrm{S}, v\rangle \quad (t,b,s,d) \vdash Y_2 \;\Rightarrow\; \langle \mathrm{S}, \mathit{cell}(l)\rangle}{(t,b,s,\delta) \vdash \texttt{store}\ Y_1\ \texttt{in}\ Y_2 \;\Rightarrow\; (\emptyset,\emptyset,\mathit{d2s}(\mathit{update}(s,l,v),[l]),\texttt{complete})}$$

$$\frac{(t,b,s,d) \vdash Y_1 \;\Rightarrow\; \langle \mathrm{D}, y_1, \_\rangle \quad (t,b,s,d) \vdash Y_2 \;\Rightarrow\; \langle \mathrm{S}, \mathit{cell}(l)\rangle}{(t,b,s,\delta) \vdash \texttt{store}\ Y_1\ \texttt{in}\ Y_2 \;\Rightarrow\; (\emptyset,\emptyset,\mathit{s2d}(s,[l]),\texttt{store}\ y_1\ \texttt{in}\ \langle \mathrm{S}, \mathit{cell}(l)\rangle \Uparrow)}$$

$$\frac{(t,b,s,d) \vdash Y_1 \;\Rightarrow\; w \quad (t,b,s,d) \vdash Y_2 \;\Rightarrow\; \langle \mathrm{D}, y_2, \_\rangle}{(t,b,s,\delta) \vdash \texttt{store}\ Y_1\ \texttt{in}\ Y_2 \;\Rightarrow\; (\emptyset,\emptyset,s,\texttt{store}\ w \Uparrow\ \texttt{in}\ y_2)}$$

Fig. 3. Primitive actions

where $A \in$ Action, $t, t' \in$ PE-Data, $b, b' \in$ PE-Binding, $s, s' \in$ Storage, δ keeps the inherited information necessary for the partial evaluation of A, and a ($\in$ Action) represents a residual action. An action which can be performed completely at partial-evaluation time is called *static action.* A static action is basically *eliminable* except where there is no enclosing action or that it is the body of abstraction. For example, `give 3` is a static action giving a static value 3, but since the result of partial evaluation should become an action, the action is not eliminable. As another example, consider an action `give abstraction of give 3`. The action `give 3` in the body of the abstraction is static, but it should be reconstructed because it is the body of the abstraction. Of course, all dynamic actions are not eliminable. An "eliminable tag" is maintained in the environment(δ) to help decide whether or not a particular action is eliminable.

The partial evaluation rules for some primitive actions are shown in Fig. 3. We call a functional-facet action *static* if it gives data which are all known, and *dynamic* otherwise. The first rule for `give` Y shows that an eliminable static action is transformed into `complete` which might be removed later when it is combined with an action combinator. Notice that the dynamic action may give data, some of whose components are static. The second rule indicates that an eliminable dynamic action is residualized, but its static components are not reconstructed. For an action which is not eliminable, both static and dynamic data should be residualized as the third rule shows.

Since bindings are forced to be statically scoped in our subset of action notation, binding actions may be performed at compile-time. For example, every constant identifier may be replaced by its constant value at compile-time, and every variable may be replaced by its relative address at compile-time. An eliminable and static binding action is transformed into `complete` (see the first rule). If an abstraction is to be bound, it is marked with "bound" so that the information can be used when an enaction of the abstraction takes place. (We will get back to this issue later again in the section.) A dynamic or non-eliminable action is residualized as the second rule shows.

In order to perform imperative actions at partial-evaluation time, great care must be taken. In a block-structured language, the `allocate a cell` action may be statically performed unless it appears in the body of `unfolding` action or dynamic branch (`then either .. or ..` combinator). The information on whether or not an action is statically allocatable during partial evaluation is maintained in the environment δ. When partially evaluated, the action `allocate a cell` is transformed into `reserve a cell and then give a cell(l)`. If the integer value stored in the cell is known, the value may be used to perform the partial evaluation. We call the cell in which a known value is stored, *static*, and otherwise *dynamic.* If a cell is static, the cell might be updated with a known value, and marked with "static" indicating the value in the cell is static. The action processing a static cell may be eliminable during partial evaluation. If every value stored in a cell is static throughout the computation, then the `reserve a cell` action does not have to be residualized, thus removed. For example, the following (not very realistic) action:

```
| allocate a cell
| and then allocate a cell
| and then allocate a cell
then
| | store 3 in the given cell#2
| and then
| | give the value stored in the cell#1
```

is transformed into:

```
| reserve a cell
then
| give the value stored in the cell(0)
```

4.4 Abstraction and enaction

The body action inside a statically-scoped abstraction, `closure abstraction of` A, is partially evaluated with completely unknown transient data, current bindings and storage. The result is a static abstraction with its body partially evaluated as shown in the first rule in Fig.4. The partially evaluated abstraction may be either given as a transient or bound to an identifier. In the former case, when it is enacted, it is merely partially evaluated with respect to available data. But in the latter case, if it is not completely enacted (which means the abstraction is specialized with respect to some known data), it is necessary to reconstruct a binding action producing the specialized abstraction. The process of partially evaluating `enact application` Y_1 `to` Y_2 is formally described in Fig.4. Let's look at a few examples. The action

```
furthermore
| bind "f" to closure abstraction of give sum(it#1,it#2)
hence
| enact application abstraction bound to "f" to (1,2)
```

is transformed to

```
give 3.
```

The action

```
furthermore
| bind "f" to closure abstraction of give sum(it#1,it#2)
hence
| enact application abstraction bound to "f"
|                 to (3,the value bound to "x")
```

is transformed to, if the value bound to `x` is unknown,

```
furthermore
| bind "f1" to closure abstraction of given sum(3,it#1)
hence
| enact application abstraction bound to "f1"
|                 to the value bound to "x".
```

$$\frac{\begin{array}{c} l \;=\; \textit{length-of-tuple-in}(\mathrm{A}) \\ \delta = [\,\textit{statically\_allocatable} \leftarrow \textit{true}, \\ \quad \textit{eliminable} \leftarrow \textit{false}, \\ \quad \textit{unfolding\_body} \leftarrow \texttt{complete}, \\ \quad \textit{depth} \leftarrow d+1, \\ \quad \textit{in\_dynamic\_branch} \leftarrow \textit{false}] \\ ((\langle \mathrm{D}, \texttt{it}\#1, 1\rangle, ..., \langle \mathrm{D}, \texttt{it}\#l, l\rangle), b, s, \delta) \vdash\; A \;\Rightarrow\; (t', b', s', a') \end{array}}{(t,b,s,d) \vdash\; \texttt{closure abstraction of}\; A \;\Rightarrow\; \langle \mathrm{S}, \textit{abstraction}(a', b, \textit{unbound})\rangle}$$

$$\frac{\begin{array}{c} (t,b,s,d) \vdash\; Y_1 \Rightarrow \langle \mathrm{S}, \textit{abstraction}(A, b', \_)\rangle \\ (t,b,s,d) \vdash\; Y_2 \Rightarrow w_2 \\ \textit{all-static}(w_2) \\ (w_2, b', s, \delta) \vdash A \Rightarrow (t_1, b_1, s_1, a_1) \end{array}}{(t,b,s,\delta) \vdash\; \texttt{enact application}\; Y_1 \;\texttt{to}\; Y_2 \;\Rightarrow\; (t_1, b_1, s_1, a_1)}$$

$$\frac{\begin{array}{c} (t,b,s,d) \vdash\; Y_1 \Rightarrow \langle \mathrm{S}, \textit{abstraction}(A, b', \textit{unbound})\rangle \\ (t,b,s,d) \vdash\; Y_2 \Rightarrow w_2 \\ \textit{not all-static}(w_2) \\ (w_2, b', s, \delta) \vdash A \Rightarrow (t_1, b_1, s_1, a_1) \end{array}}{\begin{array}{l} (t,b,s,\delta) \vdash\; \texttt{enact application}\; Y_1 \;\texttt{to}\; Y_2 \;\Rightarrow \\ (t_1, b_1, s_1, \texttt{enact application}\; \langle \mathrm{S}, \textit{abstraction}(a_1, b', \textit{unbound})\rangle \Uparrow \\ \qquad\qquad\qquad\qquad \texttt{to}\; \textit{eliminate\_S}(w_2) \Uparrow) \end{array}}$$

$$\frac{\begin{array}{c} (t,b,s,d) \vdash\; Y_1 \Rightarrow \langle \mathrm{S}, \textit{abstraction}(A, b', \textit{bound})\rangle \\ (t,b,s,d) \vdash\; Y_2 \Rightarrow w_2 \\ \textit{not all-static}(w_2) \\ (w_2, b', s, \delta) \vdash A \Rightarrow (t_1, b_1, s_1, a_1) \\ I\_w_2 \;=\; \textit{update\_stack}(I, w_2, \textit{abstraction}(a_1, b', \textit{bound})) \end{array}}{\begin{array}{l} (t,b,s,\delta) \vdash\; \texttt{enact application}\; Y_1 \;\texttt{to}\; Y_2 \;\Rightarrow \\ (t_1, b_1, s_1, (\texttt{enact application abstraction bound to}\; I\_w_2 \\ \qquad\qquad\qquad\qquad \texttt{to}\; \textit{eliminate\_S}(w_2) \Uparrow) \end{array}}$$

$$\frac{\begin{array}{c} (t,b,s,d) \vdash\; Y_1 \Rightarrow \langle \mathrm{D}, y_1, i\rangle \\ (t,b,s,d) \vdash\; Y_2 \Rightarrow w_2 \\ l = \textit{length-of-tuple-out}(\texttt{enact application}\; Y_1 \;\texttt{to}\; Y_2) \end{array}}{\begin{array}{l} (t,b,s,\delta) \vdash\; \texttt{enact application}\; Y_1 \;\texttt{to}\; Y_2 \;\Rightarrow \\ (\langle \mathrm{D}, \texttt{it}\#1\rangle, ..., \langle \mathrm{D}, \texttt{it}\#l\rangle, \emptyset, s, (\texttt{enact application}\; \langle \mathrm{D}, y_1, i\rangle \Uparrow \;\texttt{to}\; w_2 \Uparrow)) \end{array}}$$

Fig. 4. Abstraction and enaction

$$
\frac{\begin{array}{c}
e \;=\; \text{if } (A_2 = \texttt{unfold} \;\text{ and }\; \mathit{in\_dynamic\_branch}(\delta) = \mathit{true}) \text{ then } \mathit{false} \\
\delta_1 \;=\; \delta[\mathit{eliminable} \leftarrow e] \\
(t, b, s, \delta_1) \vdash\; A_1 \;\Rightarrow\; (t_1, b_1, s_1, a_1) \\
(t_1, b, s_1, \delta) \vdash\; A_2 \;\Rightarrow\; (t_2, b_2, s_2, a_2)
\end{array}}{\begin{array}{l}
(t, b, s, \delta) \vdash\; A_1 \;\texttt{then}\; A_2 \;\Rightarrow\; (\mathit{rearrange}(t_2), b_2, s_2, a_3) \\
\qquad \mathit{where}\; a_3 = \mathit{case}\; (a_1, a_2)\; \mathit{of} \\
\qquad\qquad\quad (\texttt{complete}, \_\,) \;\rightarrow\; a_2 \\
\qquad\qquad\;\mid\; (\_\,, \texttt{complete}) \;\rightarrow\; \texttt{complete} \\
\qquad\qquad\;\mid\; (\_\,, \_\,) \;\rightarrow\; a_1 \;\texttt{then}\; a_2
\end{array}}
$$

$$
\frac{\begin{array}{c}
e \;=\; \text{if } (A_2 = \texttt{unfold} \;\text{ and }\; \mathit{in\_dynamic\_branch}(\delta) = \mathit{true}) \text{ then } \mathit{false} \\
\delta_1 \;=\; \delta[\mathit{eliminable} \leftarrow e] \\
(t, b, s, \delta_1) \vdash\; A_1 \;\Rightarrow\; (t_1, b_1, s_1, a_1) \\
(t, b, s_1, \delta) \vdash\; A_2 \;\Rightarrow\; (t_2, b_2, s_2, a_2)
\end{array}}{\begin{array}{l}
(t, b, s, \delta) \vdash\; A_1 \;\texttt{and then}\; A_2 \;\Rightarrow\; (\mathit{rearrange}(t_1, t_2), \mathit{disjoint}(b_1, b_2), s_2, a_3) \\
\qquad \mathit{where}\;\; a_3 = \mathit{case}\; (a_1, a_2)\; \mathit{of} \\
\qquad\qquad\quad (\texttt{complete}, \_\,) \;\rightarrow\; a_2 \\
\qquad\qquad\;\mid\; (\_\,, \texttt{complete}) \;\rightarrow\; a_1 \\
\qquad\qquad\;\mid\; (\_\,, \_\,) \;\rightarrow\; a_1 \;\texttt{and then}\; a_2
\end{array}}
$$

$$
\frac{\begin{array}{c}
e \;=\; \text{if } (A_2 = \texttt{unfold} \;\text{ and }\; \mathit{in\_dynamic\_branch}(\delta) = \mathit{true}) \text{ then } \mathit{false} \\
\delta_1 \;=\; \delta[\mathit{eliminable} \leftarrow e] \\
(t, b, s, \delta_1) \vdash\; A_1 \;\Rightarrow\; (t_1, b_1, s_1, a_1) \\
\delta_2 \;=\; \mathit{increase\_depth}(\delta) \\
(t, b@b_1, s_1, \delta_2) \vdash\; A_2 \;\Rightarrow\; (t_2, b_2, s_2, a_2) \\
b\_a \;=\; \mathit{residualize\_specialized\_abstraction}()
\end{array}}{\begin{array}{l}
(t, b, s, \delta) \vdash\; \texttt{furthermore}\; A_1 \;\texttt{hence}\; A_2 \;\Rightarrow\; (\mathit{rearrange}(t_2), b_2, s_2, a_3) \\
\qquad \mathit{where}\; a_3 = \mathit{case}\; (a_1, b\_a, a_2)\; \mathit{of} \\
\qquad\qquad\quad (\_\,, \_\,, \texttt{complete}) \;\rightarrow\; \texttt{complete} \\
\qquad\qquad\;\mid\; (\texttt{complete}, \texttt{complete}, \_\,) \;\rightarrow\; a_2 \\
\qquad\qquad\;\mid\; (\texttt{complete}, \_\,, \_\,) \;\rightarrow\; \texttt{furthermore}\; b\_a \;\texttt{hence}\; a_2 \\
\qquad\qquad\;\mid\; (\_\,, \texttt{complete}, \_\,) \;\rightarrow\; \texttt{furthermore}\; a_1 \;\texttt{hence}\; a_2 \\
\qquad\qquad\;\mid\; (\_\,, \_\,, \_\,) \;\rightarrow\; \texttt{furthermore}\; a_1 \;\texttt{before}\; b\_a \;\texttt{hence}\; a_2
\end{array}}
$$

$$
\frac{\begin{array}{c}
e \;=\; \text{if } (A_2 = \texttt{unfold} \;\text{ and }\; \mathit{in\_dynamic\_branch}(\delta) = \mathit{true}) \text{ then } \mathit{false} \\
\delta_1 \;=\; \delta[\mathit{eliminable} \leftarrow e] \\
(t, b, s, \delta_1) \vdash\; A_1 \;\Rightarrow\; (t_1, b_1, s_1, a_1) \\
(t, b@b_1, s_1, \delta) \vdash\; A_2 \;\Rightarrow\; (t_2, b_2, s_2, a_2)
\end{array}}{\begin{array}{l}
(t, b, s, \delta) \vdash\; A_1 \;\texttt{before}\; A_2 \;\Rightarrow\; (\mathit{rearrange}(t_2), b@b_1@b_2, s_2, a_3) \\
\qquad \mathit{where}\; a_3 = \mathit{case}\; (a_1, a_2)\; \mathit{of} \\
\qquad\qquad\quad (\texttt{complete}, \_\,) \;\rightarrow\; a_2 \\
\qquad\qquad\;\mid\; (\_\,, \texttt{complete}) \;\rightarrow\; a_1 \\
\qquad\qquad\;\mid\; (\_\,, \_\,) \;\rightarrow\; a_1 \;\texttt{before}\; a_2
\end{array}}
$$

Fig. 5. Action combinators

4.5 Action combinators

Fig.5 describes the partial evaluation of action combinators. In the action combinator A_1 `then` A_2, when A_1 is static, A_1 is eliminable because the data given by A_1 is not alive anymore outside A_1 `then` A_2. In the action combinator A_1 `and then` A_2, both A_1 and A_2 are partially evaluated sequentially, and then the results are combined in a proper way. Since the components of resulting transient tuple can either be static or dynamic, they should be rearranged to have only dynamic values and the index should be changed accordingly. In the action `furthermore` A_1 `hence` A_2, when A_1 is static, A_1 is eliminable because the bindings produced by A_1 is not alive anymore outside A_2. As we explained in the previous subsection on enaction, after the partial evaluation of A_2, the specialized abstraction-binding actions are reconstructed. In the action A_1 `before` A_2, A_1 and A_2 are partially evaluated in turn, and then the results are combined accordingly. In all of these four action combinators, if the right subaction A_2 is `unfold` and the whole action is in a dynamic branch, then when the left subaction A_1 is partially evaluated, the eliminable tag should be set to *false* so that A_1 is not eliminated even if it is static. The details are formally described in Fig.5.

$$\frac{\begin{array}{c}\delta_1 \;=\; \delta[\mathit{statically\_allocatable} \leftarrow \mathit{false}, \mathit{unfolding\_body} \leftarrow A] \\ (t,b,s,\delta_1) \vdash \; A \;\Rightarrow\; (t_1,b_1,s_1,a_1)\end{array}}{(t,b,s,\delta) \vdash \; \texttt{unfolding}\; A \;\Rightarrow\; (t_1,b_1,s_1,a_1)}$$

$$\frac{\begin{array}{c}\mathit{not}\; \mathit{in\_dynamic\_branch}(\delta) \\ (t,b,s,\delta) \vdash \; \mathit{unfolding\_body}(\delta) \;\Rightarrow\; (t_1,b_1,s_1,a_1)\end{array}}{(t,b,s,\delta) \vdash \; \texttt{unfold} \;\Rightarrow\; (t_1,b_1,s_1,a_1)}$$

$$\frac{\begin{array}{c}\mathit{in\_dynamic\_branch}(\delta) \\ (t,b,s,\delta) \overset{\sim}{\vdash} \mathit{unfolding\_body}(\delta) \;\Rightarrow\; (t_1,b_1,s_1,a_1)\end{array}}{(t,b,s,\delta) \vdash \; \texttt{unfold} \;\Rightarrow\; (t_1,b_1,s_1,\mathit{push}(a_1))}$$

Fig. 6. Unfolding action

The unfolding action retains the termination problem of online partial evaluation, meaning that if the original action does not terminate, it is not guaranteed the partial evaluation terminates. Another problem is the code explosion. We do not deal with this problem in detail in this paper. The current version of our partial evaluator keeps unfolding if it is possible to unfold. If the `unfold` action is in one of the dynamic branches, the body of the `unfolding` action is partially evaluated just once. As shown in Fig.6, a static allocation is not allowed in the body of an unfolding action.

$$\frac{(t,b,s,d) \vdash\ Y\ \Rightarrow\ \langle \mathrm{S}, \mathit{true} \rangle \qquad (t,b,s,\delta) \vdash\ A_1\ \Rightarrow\ (t_1,b_1,s_1,a_1)}{(t,b,s,\delta) \vdash\ Y\ \texttt{then either}\ A_1\ \texttt{or}\ A_2\ \Rightarrow\ (t_1,b_1,s_1,a_1)}$$

$$\frac{(t,b,s,d) \vdash\ Y\ \Rightarrow\ \langle \mathrm{S}, \mathit{false} \rangle \qquad (t,b,s,\delta) \vdash\ A_2\ \Rightarrow\ (t_2,b_2,s_2,a_2)}{(t,b,s,\delta) \vdash\ Y\ \texttt{then either}\ A_1\ \texttt{or}\ A_2\ \Rightarrow\ (t_2,b_2,s_2,a_2)}$$

$$\frac{\begin{array}{c}(t,b,s,d) \vdash\ Y\ \Rightarrow\ \langle \mathrm{D}, y, \_ \rangle \\ l\ =\ \mathit{length\text{-}of\text{-}tuple\text{-}out}(Y\ \texttt{then either}\ A_1\ \texttt{or}\ A_2) \\ \delta_1\ =\ \delta[\,\mathit{statically\_allocatable} \leftarrow \mathit{false}, \\ \qquad \mathit{in\_dynamic\_branch} \leftarrow \mathit{true}] \\ (t,b,s,\delta_1) \vdash\ A_1\ \Rightarrow\ (t_1,b_1,s_1,a_1) \\ (t,b,s,\delta_1) \vdash\ A_2\ \Rightarrow\ (t_2,b_2,s_2,a_2)\end{array}}{\begin{array}{c}(t,b,s,\delta) \vdash\ Y\ \texttt{then either}\ A_1\ \texttt{or}\ A_2\ \Rightarrow \\ ((\langle \mathrm{D}, \texttt{it\#}1, 1\rangle, ..., \langle \mathrm{D}, \texttt{it\#}l, l\rangle), \emptyset, s^*, (y\ \texttt{then either}\ a_1''\ \texttt{or}\ a_2''))\end{array}}$$

where
- $s^* = s_1' \sqcup s_2'$
- $(a_1', s_1') = B(s_1, s_2)$
  $(a_2', s_2') = B(s_2, s_1)$
- $B(s_1, s_2) =$

```
  Find all cell(i)'s in s_1 that satisfies the following condition:
    the cell(i) in s_1 is static and
    (the values of cell(i) in s_1 and s_2 are different  or
     the cell(i) in s_2 dynamic)
  Let them be cell(i_1), ..., cell(i_m)
  Then
    (store n_1 in cell(i_1) and then ··· and then store n_m in cell(i_m), s')
    where n_1 = slookup(i_1, s_1)
          n_2 = slookup(i_2, s_1)
          ···
          n_m = slookup(i_m, s_1)
          s' = s2d(s_1, [i_1, ..., i_m]) {mark cell(i_1), ..., cell(i_m) all dynamic}
```

- if $\mathit{there\_is\_unfold\_in}(a_1)$ then
  $a_1'' = a_1[\ a_1'\ \texttt{and then unfolding}\ \mathit{pop}()/\texttt{unfold}\ ]$
  else
  $a_1'' = a_1\ \texttt{and then}\ a_1'$
- if $\mathit{there\_is\_unfold\_in}(a_2)$ then
  $a_2'' = a_2[\ a_2'\ \texttt{and then unfolding}\ \mathit{pop}()/\texttt{unfold}\ ]$
  else
  $a_2'' = a_2\ \texttt{and then}\ a_2'$

Fig. 7. Branch combinators

As shown in Fig.7, in the action Y `then either` A_1 `or` A_2, if Y is static, then according to the yielded truth value, either A_1 or A_2 is partially evaluated. If Y is dynamic, the followings should be taken into account:

1. It is impossible to allocate a cell at partial-evaluation time.
2. If it appears in the body of `unfolding` action and the `unfold` action appears either in A_1 or in A_2, the `unfolding` action must not be partially evaluated since it causes an infinite unfolding.
3. Even if any of A_1 and A_2 are static, they should be residualized because one of them may be used by enclosing action. The `store` Y_1 `in` Y_2 action is residualized if different branches alter the same cell with different values. This particular residual action is known as an explicator in partial evaluation terminology [5]. The details on the explicator actions are specified in Fig.7.
4. The resulting stores of partially evaluating both branches should be joined because it is impossible to know at partial-evaluation time which branch will be executed at run time.

5 Implementation

The system presented in this paper has been implemented in Standard ML. We have collected and tested four action programs including three test programs used in the ACTRESS system [1]. The table below compares the action-expansion time with the partial-evaluation time (including expansion time), and the interpreting time of naively expanded action programs with that of partially evaluated ones. The system used is the Sun SPARC Station under Sun OS 5.5.

Program	Compile Time		Running Time		
	normal	with P.E.	normal	with P.E.	advantage
loop	0.39	11.9	5.37	0.30	17.9
binding	0.33	1.02	0.73	0.30	2.43
loopfact	0.32	10.00	3.37	0.30	11.23
looppower	0.32	4.61	1.61	0.90	1.79

6 Related works

Moura and Watt first identified and formalized action transformations by defining transformation rules and algebraic laws [9] for their ACTRESS compiler generation system [1].

Doh and Schmidt also employed the static evaluation (constant propagation and dead code removal) of action denotation using the binding-time information in their action-semantics prototyping system [3].

In Doh's previous work [2], action transformations were carried out by off-line partial evaluation using two-level type system. The transformations achieved by his system include transient and binding elimination. However, his off-line method cannot partially evaluate actions inside the body of unfolding-action and abstraction without performing separate global analysis.

7 Conclusion

We have presented an automatic action-transformation method based on online partial evaluation. Not like the previous off-line method, we achieved the partial evaluation of actions inside the body of unfolding-action and abstraction using online approach. In addition to the transient and binding elimination, we extended the method so that imperative actions can be partially evaluated.

Our future investigation will be on proving the correctness of our online partial evaluation and on improving the performance of the current online partial evaluator for actions. However, it is not quite certain at this point of time that the full-powered partial evaluation of actions is actually beneficial to action-semantics-directed compiler generators. If we see actions as intermediate code in compilers, there might be some trade-off on how much needs to be partially evaluated. It could be enough to do merely some constant folding and binding elimination just for compiling purposes. This leaves us more experiments to be done.

References

1. Deryk F. Brown, Hermano Moura, and David A. Watt. ACTRESS: an action semantics directed compiler generator. In *CC'92, Proceedings of the 4th International Conference on Compiler Construction, Paderborn*, Lecture Notes in Computer Science 641, pages 95–109. Springer-Verlag, 1992.
2. Kyung-Goo Doh. Action transformation by partial evaluation. In *PEPM'95, Proceedings of Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 230–240. ACM, June 1995.
3. Kyung-Goo Doh and David A. Schmidt. Action semantics-directed prototyping. *Computer Languages*, 19(4):213–233, 1993.
4. Kent D. Lee. *Action Semantics-based Compiler Generation.* PhD thesis, University of Iowa, 1999.
5. Uwe Meyer. Techniques for partial evalution of imperative languages. In *PEPM'91, Proceedings of Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 94–105. ACM, June 1991.
6. Peter D. Mosses. *Action Semantics.* Cambridge Tracts in Theoretical Computer Science 26. Cambridge University Press, 1992.
7. Peter D. Mosses. A tutorial on action semantics. In *Notes for FME'94, Formal Methods Europe, Barcelona*, October 24–28 1994.
8. Hermano Moura. *Action Notation Transformations.* PhD thesis, University of Glasgow, 1993.

9. Hermano Moura and David A. Watt. Action transformations in the ACTRESS compiler generator. In Peter Fritzon, editor, *CC'94, Proceedings of the 5th International Conference on Compiler Construction , Edinburgh*, Lecture Notes in Computer Science 786, pages 1–15. Springer-Verlag, April 1994.
10. Carsten K. Gomard Neil D. Jones and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall International, 1993.
11. Peter Ørbæk. OASIS: An Optimizing Action-based Compiler Generator. In Peter Fritzon, editor, *CC'94, Proceedings of the 5th International Conference on Compiler Construction , Edinburgh*, Lecture Notes in Computer Science 786, pages 16–30. Springer-Verlag, April 1994.
12. Jens Palsberg. *Provably Correct Compiler Generation.* PhD thesis, Aarhus University, 1992.
13. David A. Watt. *Programming Language Syntax and Semantics.* Prentice-Hall International, Englewood Cliffs, New Jersey, 1991.

An Algebra of Actions

S. B. Lassen*

University of Cambridge Computer Laboratory
Soeren.Lassen@cl.cam.ac.uk

Abstract. A long-standing challenge concerning the foundations of action semantics is the development of the algebraic theory of *actions*, the computational entities used in action-semantic descriptions of programming languages. In the development of action semantics, much emphasis has been placed on the algebraic laws that are to be satisfied by the combinators and primitives of the action notation. Yet, because of the size and certain features of the notation, the task of developing its algebraic theory appears insurmountable. Towards circumventing these difficulties, this paper introduces a new core algebra of actions: a smaller core notation of actions with more algebraic laws and equipped with a syntactic reduction semantics. The core algebra should be more amenable to theoretical analysis. It is designed to facilitate a direct mapping of almost all existing action notation into it, so it may serve as an interface for theoretical work on action semantics.

1 Introduction

Action semantics (`www.brics.dk/Projects/AS`) is a formal semantics framework with good pragmatic properties. These derive from the design of *action notation*, the metalanguage for expressing the semantic entities, called *actions*, used in action-semantic descriptions of programming languages. Action notation is a rather large combinator-based language with a range of primitives and combinators for directly specifying many common programming language concepts, including under-specified evaluation order, imperative operations on an underlying store, and asynchronous processes and message-passing. This makes it ideally suited for directly describing most programming language constructs.

The syntax and semantics of action notation are defined in [Mos92] and, recently, a new, modular structural operational semantics has been specified [Mos99]. However, the size of the notation and certain aspects of its design make formal reasoning about actions cumbersome. A set of equational laws are specified in [Mos92, App. B] and laws for a substantial functional fragment have been verified operationally in [Las97]. Rather few equational laws are specified for the remaining notation and these have not been formally verified. Mosses [Mos92,Mos96] has repeatedly stressed the need for further development of the algebraic theory of actions and theories for action semantics reasoning about programs. In particular, he has asked for a useful proof calculus for the concurrent and communicative part of action notation.

* Supported by grant number GR/L38356 from the UK EPSRC.

This paper presents a compact core algebra of actions which should be better suited for formal reasoning, in particular equational reasoning. The algebra is basically an extension of the abstract semantic algebras in [Mos83], a precursor to action semantics. The operational behaviour of the core actions is specified by a reduction semantics. The core algebra includes a new construct that allows flexible wiring of the information flow, supporting formulations of many laws involving information flow, including (1) laws that reduce the many sequencing combinators of action notation to one basic sequencing combinator and (2) new expansion laws that reduce interleaving to non-determinism. A novel, uniform syntactic treatment of storage cells, spawned processes, and sent messages facilitates the formulation of reduction rules and algebraic laws for imperative, concurrent and communicating actions.

The core algebra is designed to facilitate a direct mapping of almost all existing action notation into it. Foremost, the algebra is meant as an interface for theoretical work on action semantics, but it also suggests potential simplifications to standard action notation.

Much work remains to be done. First of all, the consistency of the algebraic theory should be verified. The algebraic laws have been formulated with a form of testing equivalence [DH84] in mind but, regardless, it is likely to be a formidable task to provide verification techniques for all facets of the algebra. Another interesting challenge is whether our expansion laws can contribute to an equational completeness result for any worthwhile fragment of the algebra.

The paper is organised as follows. In §2 the syntax of the core algebra is defined and compared with standard action notation. The reduction semantics and algebraic properties of the core primitives and combinators are presented step-wise in §3–§6, roughly structured according to the standard facets of action notation. Then §7 discusses the interleaving of control flow and gives expansion laws for reducing interleaving to non-determinism. Finally, §8 takes stock of the core algebra.

The reader is assumed to be familiar with action notation.

Relations The paper uses the following notations for relations between action terms. Firstly, $\Phi \equiv \Phi'$ denotes syntactic equality between syntactic phrases Φ and Φ'. The relations $=$ and $\sqsubseteq$ are, respectively, the semantic equivalence and semantic preorder on actions defined by the equational and inequational laws of the algebra. A finer structural congruence relation between actions terms is denoted by $\rightleftharpoons$. The reduction relation $\rightarrow$ is also a relation between action terms. The labelled transition relation, written $A, \vec{O} \xrightarrow{\rho} A', \vec{O}'$, is between configurations consisting of an action term paired with a sequence of objects, and is labelled by a renaming ρ of any references to the objects in $\vec{O}$.

2 The notation

The abstract syntax of the actions of the core algebra is defined as a syntactic sort *Action* specified by the grammar in Table 1 together with auxiliary syntactic sorts *Object*, *Informer*, *token*, and *data*.

$$
\begin{array}{lll}
\textit{Action} & A ::= & I \mid \mathsf{raise} \mid \mathsf{unfolding}\, A \mid \mathsf{unfold}\#p \qquad\qquad\qquad (p \geq 1) \\
 & & \mid A\ \mathsf{or}\ A \mid A > A \setminus A \mid A\ \mathsf{and}\ A \mid \mathsf{indivisibly}\, A \\
 & & \mid A \circ I \mid \mathsf{find}\, t \mid (\textit{dataop}) \mid \mathsf{enact} \mid \mathsf{copying}\, A \\
 & & \mid A\ \mathsf{with\ new}\ \vec{O} \mid \mathsf{update} \mid \mathsf{access} \mid \mathsf{receive} \mid \mathsf{compare} \\
\textit{Object} & O ::= & \mathsf{cell}\, I \mid \mathsf{agent}\, A \mid \mathsf{message}\, I \\
\textit{Informer} & I ::= & \mathsf{complete} \mid \mathsf{skip} \mid I\ \mathsf{and}\ I \mid I \circ I \mid \mathsf{rebind} \mid \mathsf{bind}\, t \mid \mathsf{regive} \mid d \\
 & & \mid \mathsf{copying}\, I \mid \mathsf{copy}\#p \\
\textit{token} & t ::= & \ldots \quad (\textit{e.g., alphanumeric text strings}) \\
\textit{data} & d ::= & () \mid (d, d) \mid \mathsf{abstraction\ of}\ A \mid \mathsf{cell}\#p \mid \mathsf{agent}\#p \mid \mathsf{self}\#p \mid \ldots
\end{array}
$$

$\vec{O}$ denotes a finite, possibly empty sequence of objects $O_1, \ldots, O_n$ $(n \geq 0)$

Table 1. Syntax

The sort *data* is open-ended: it may include application-specific data types, including compound data built from abstractions (abstraction of A) and object references (cell$\#p$, agent$\#p$, and self$\#p$). Similarly, *dataop* ranges over application-specific data operations.

The remainder of this section gives a quick introduction to the algebra by highlighting the most important differences compared with action notation.

- The $\#p$ suffixes in unfold$\#p$, copy$\#p$, and object references are de Bruijn indices with static scope, referring to the p'th lexically enclosing combinator of the appropriate kind (unfolding, copying, with new, or agent /abstraction of). The scope of these combinators extends into abstractions occurring in data constants in their scope. The indices are optional; the default is #1. For instance, unfold abbreviates unfold#1 and has the same meaning as in action notation.
- There are only two kinds of termination, namely by completing and by escaping; skip completes with the incoming information flow as outcome and, likewise, raise escapes with all incoming information flow (for uniformity, both functional and declarative information, not just functional information as the escape primitive in action notation). The notions of failure and commitment in action notation are omitted. Instead the exception mechanism is intended to express error reporting, error handling and branching.
- The ternary sequencing combinator $A_0 > A_1 \setminus A_2$ combines sequencing and exception handling. It is the only sequencing combinator in the core algebra.

In conjunction with the copying and copy$\#p$ constructs that allow flexible wiring of information flow, all other standard sequencing combinators may be encoded.
– In place of yielders there is the slightly different notion of informers. They are more restricted than yielders in that they must act as total functions on the incoming information flow—they must complete, they may not access the state, and they cannot even look up bindings because this is a partial operation as tokens may be unbound.
– For every data operation *dataop*, (*dataop*) is the action which applies *dataop* to the incoming functional information flow and completes with the result on the functional data flow, if defined, otherwise it raises an exception. Every data constant d is a completing action with outcome d. The compare action compares object references for equality.
– The directive facet for specifying circular bindings is omitted, in part because the core algebra provides enhanced facilities for specifying recursive structures, and in part because redirections, when needed, can be adequately represented by means of cells from the imperative facet.
– The store model is in the style of Standard ML: memory cells are always initialised and cannot be (explicitly) deallocated, and there is no primitive corresponding to action notation's current storage yielder for reading out the contents of all memory cells.
– The creation and referencing of the different kinds of state objects—cells, messages, and agents—are treated uniformly. The A with new $\vec{O}$ construction creates a sequence of possibly mutually recursive objects, $\vec{O}$ before performing A. The objects in $\vec{O}$ are referenced by means of references of the form cell$\#p$ and agent$\#p$, and lexically enclosing agents are referenced by self$\#p$. In this formalism, cells are naturally shared between agents (following [MM93] rather than standard action notation).

All in all, the constructs of the core notation are designed to support the formulation of equational laws and symbolic computation of actions, and they appear to offer a convenient expressive power practically equivalent to the existing action notation. The notation of the core algebra is about half the size of the kernel action notation in [Mos92, App. C].

3 The basic facet

The basic facet of the core algebra consists of primitives and combinators for specifying the flow of control in actions. They are given by the grammar:

$$\begin{array}{rcl} \textit{Action}\ A & ::= & \mathsf{skip} \mid \mathsf{raise} \mid \mathsf{unfolding}\, A \mid \mathsf{unfold}\#p \qquad (p \geq 1) \\ & & \mid A \,\mathsf{or}\, A \mid A > A \setminus A \mid A \,\mathsf{and}\, A \mid \mathsf{indivisibly}\, A \end{array}$$

Actions either complete, escape, or diverge, or choose non-deterministically between two or more of these termination behaviours. The skip primitive completes and raise escapes. (These two basic primitives are not called complete and escape as in action notation because they behave differently with regard to information flow in the next section.)

The $\mathsf{unfold}\#p$ primitive marks a recursive unfolding of the p'th innermost lexically enclosing unfolding combinator, i.e., $\#p$ is a de Bruijn index. The indexed unfolds facilitate the coding of nested recursion. (Although it is not obvious that this generality is really needed, indexed unfolds are included here in analogy with several other indexed reference constructs in the remainder of the core algebra.)

There are no notions of failure and commitment in the core algebra. Hence the or choice combinator is just an erratic choice. This simplification compared to action notation appears to be useful towards obtaining a strong algebraic theory: although the or combinator is also associative, commutative and idempotent in action notation, its behaviour with regard to failure and commitment leads to complex interactions with the other language constructs which are difficult to capture equationally.

The ternary combinator $(A > A_1 \setminus A_2)$ first performs A whereupon it forwards control to A_1 if A completes, and to A_2 if A escapes.

The and and indivisibly combinators have their usual meaning from action notation: $(A_1 \;\mathsf{and}\; A_2)$ interleaves the performance of A_1 with that of A_2, and $(\mathsf{indivisibly}\, A)$ performs A as one atomic step not interleaved with anything else.

3.1 Substitution

Several kinds of references other than unfold appear in the full algebra. For each reference kind *ref* we define some standard notation for expressing and manipulating de Bruijn indices:

$$\begin{aligned}
\uparrow_{\mathit{ref}} &\triangleq \{\mathit{ref}\#p := \mathit{ref}\#p{+}1 \mid p \geq 1\} \\
\lceil \mathit{ref} := A \rceil &\triangleq \{\mathit{ref}\#1 := A, \mathit{ref}\#p{+}1 := \mathit{ref}\#p \mid p \geq 1\} \\
\downarrow_{\mathit{ref}} &\triangleq \lceil \mathit{ref} := \mathit{ref}\#1 \rceil
\end{aligned}$$

Notation $\Phi\{\mathit{ref}\#p := A_p \mid p \geq 1\}$ denotes the simultaneous substitution of actions $A_1, A_2, \ldots$ for references $\mathit{ref}\#1, \mathit{ref}\#2, \ldots$ in the syntactic phrase Φ, with indices suitably adjusted as substitution enters under binders. In the case of unfold, the definition is:

$$\begin{aligned}
&(\mathsf{unfold}\#p)\{\mathsf{unfold}\#p := A_p \mid p \geq 1\} \triangleq A_p \\
&(\mathsf{unfolding}\, A)\{\mathsf{unfold}\#p := A_p \mid p \geq 1\} \triangleq \\
&\qquad \mathsf{unfolding}\,(A\{\mathsf{unfold}\#1 := \mathsf{unfold}\#1, \mathsf{unfold}\#p{+}1 := A_p\uparrow_{\mathsf{unfold}} \mid p \geq 1\})
\end{aligned}$$

and for all other action constructors θ of arity $|\theta| \geq 0$,

$$\begin{aligned}
&(\theta(\Phi_1 \ldots \Phi_{|\theta|}))\{\mathsf{unfold}\#p := A_p \mid p \geq 1\} \triangleq \\
&\qquad \theta((\Phi_1\{\mathsf{unfold}\#p := A_p \mid p \geq 1\}) \ldots (\Phi_{|\theta|}\{\mathsf{unfold}\#p := A_p \mid p \geq 1\}))
\end{aligned}$$

3.2 Reduction semantics

We specify the operational behaviour of actions by a binary reduction relation, $\to$, between action terms (i.e., terms of the free term algebra, not quotiented by the ensuing equational laws). It is defined inductively by the following actions and rules.

(1) $A_1 \;\textsf{or}\; A_2 \to A_1\,; \quad A_1 \;\textsf{or}\; A_2 \to A_2$

(2) $\textsf{unfolding}\; A \to A\lceil\textsf{unfold} := \textsf{unfolding}\; A\rceil$

(3) $\textsf{skip} > A_1 \setminus A_2 \to A_1\,; \quad \textsf{raise} > A_1 \setminus A_2 \to A_2$

(4) $\textsf{raise and}\; A \to \textsf{raise}\,; \quad A \;\textsf{and raise} \to \textsf{raise}$

(5) $$\frac{A \to A'}{A > A_1 \setminus A_2 \to A' > A_1 \setminus A_2}$$

(6) $$\frac{A_1 \to A_1'}{A_1 \;\textsf{and}\; A_2 \to A_1' \;\textsf{and}\; A_2}\,; \qquad \frac{A_2 \to A_2'}{A_1 \;\textsf{and}\; A_2 \to A_1 \;\textsf{and}\; A_2'}$$

(7) $$\frac{A \to^* T}{\textsf{indivisibly}\; A \to T}$$

where $\to^*$ denotes the reflexive, transitive closure of $\to$, and T ranges over irreducible actions generated by the grammar:

$$\textit{Terminated}\;\; T \;::=\; \textsf{skip and} \ldots \textsf{and skip} \;\mid\; \textsf{raise}$$

Let evaluation contexts, E, be action terms with a hole, $[\,]$, at redex position. They are generated by the grammar:

$$\textit{Evaluation context}\;\; E \;::=\; [\,] \;\mid\; E > A \setminus A \;\mid\; E \;\textsf{and}\; A \;\mid\; A \;\textsf{and}\; E$$

Notation $E[A]$ denotes the term obtained from E be replacing its hole by A. The three reduction rules in (5)–(6) express that reduction is preserved by evaluation contexts: $A \to A'$ implies $E[A] \to E[A']$.

3.3 Equations

The equational theory of the algebra is specified by a set of axioms, or 'laws'. It is defined inductively as the smallest congruent equational theory which contains all instances of the laws.

The following laws specify algebraic properties of the basic action combinators and primitives.

(8) $\textsf{unfolding}\; A = A\lceil\textsf{unfold} := \textsf{unfolding}\; A\rceil$

(9) $A \;\textsf{or}\; A = A\,; \quad (A \;\textsf{or}\; A') \;\textsf{or}\; A'' = A \;\textsf{or}\; (A' \;\textsf{or}\; A'')\,; \quad A \;\textsf{or}\; A' = A' \;\textsf{or}\; A$

(10) $\textsf{skip} > A \setminus A' = \textsf{raise} > A' \setminus A = A = A > \textsf{skip} \setminus \textsf{raise}$

(11) $(A > A_1 \setminus A_2) > A_1' \setminus A_2' = A > (A_1 > A_1' \setminus A_2') \setminus (A_2 > A_1' \setminus A_2')$

(12) $(A \text{ or } A') > A_1 \setminus A_2 = (A > A_1 \setminus A_2) \text{ or } (A' > A_1 \setminus A_2)$

(13) $A > (A_1 \text{ or } A_1') \setminus A_2 = (A > A_1 \setminus A_2) \text{ or } (A > A_1' \setminus A_2)$

(14) $A > A_1 \setminus (A_2 \text{ or } A_2') = (A > A_1 \setminus A_2) \text{ or } (A > A_1 \setminus A_2')$

(15) $\text{skip and raise} = \text{raise and skip} = \text{raise and raise} = \text{raise}$

(16) $(A \text{ and } A') \text{ and } A'' = A \text{ and } (A' \text{ and } A'')$

(17) $(A \text{ or } A') \text{ and } A'' = (A \text{ and } A'') \text{ or } (A' \text{ and } A'')$

(18) $A \text{ and } (A' \text{ or } A'') = (A \text{ and } A') \text{ or } (A \text{ and } A'')$

(19) $\text{indivisibly indivisibly } A = \text{indivisibly } A\,; \quad \text{indivisibly } T = T$

(20) $\text{indivisibly } (A \text{ or } A') = (\text{indivisibly } A) \text{ or } (\text{indivisibly } A')$

(21) $\text{indivisibly } ((\text{indivisibly } A) > A_1 \setminus A_2) = \text{indivisibly } (A > A_1 \setminus A_2)$

(22) $\text{indivisibly } (A > (\text{indivisibly } A_1) \setminus A_2) = \text{indivisibly } (A > A_1 \setminus A_2)$

(23) $\text{indivisibly } (A > A_1 \setminus (\text{indivisibly } A_2)) = \text{indivisibly } (A > A_1 \setminus A_2)$

Remark 1. The laws for indivisibly anticipate the introduction of state. In the absence of state, indivisibly is redundant, so

(24)† $\text{indivisibly } A = A$

holds, rendering the above algebraic laws for indivisibly trivial.

3.4 Inequations

The erratic choice combinator or induces a partial order on the equational algebra, by taking or to be the greatest lower bound operator. That is, we define a relation $\sqsubseteq$ between actions as follows.

$$A \sqsubseteq A' \overset{\text{def}}{\Leftrightarrow} A = (A \text{ or } A')$$

It is a refinement relation, ordering less deterministic actions below more deterministic ones. We are going to use $\sqsubseteq$ for specifying a number of inequational laws. By the definition of $\sqsubseteq$, these may all be read as equational laws.

Because or is idempotent, associative, and commutative, the relation $\sqsubseteq$ is a partial order: reflexive, transitive, and anti-symmetric. Furthermore, the next rule specifies that $\sqsubseteq$ is compatible with respect to every action combinator θ with arity $|\theta| \geq 1$.

(25) $\dfrac{A_1 \sqsubseteq A_1' \quad \cdots \quad A_n \sqsubseteq A_n'}{\theta(A_1 \ldots A_n) \sqsubseteq \theta(A_1' \ldots A_n')} \quad (|\theta| = n)$

Some reductions $A \to A'$ resolve non-determinism so that the possible outcomes of A' are fewer than those of A. These cannot be equational laws but every reduction may be read as an inequational law:

(26) $A \sqsubseteq A', \;\text{ if } A \to A'$

4 Information flow

Actions process several kinds of information and are structured into so-called facets according to the information being processed. In the core algebra the different kinds of information are grouped under two main headings. One is the *information flow* of the functional and declarative facets—this information consists of associative tuples of data and bindings; they can be copied and discarded. The other is the *state* of the imperative and communicative facets—this information consists of sets of persistent objects: cells, agents, and messages.

In this section the algebra is first extended with 'generic' information flow primitives which make only very general assumptions about the information that is flowing. Next, the two actual kinds of information, transient data and bindings, are introduced as the functional and declarative facets of the algebra. Finally, a new copying construct is added to facilitate the encoding of action notation's many sequencing combinators.

Our general assumptions about the information flow are (1) that there is an associative binary operation for merging information, and (2) that there is a notion of empty information which is unit for merge. The complete primitive completes with the empty information. The merge operation is expressed by means of the and combinator.

To obtain a syntactic representation of information, we introduce a new syntactic sort, *Informer*, consisting of deterministic, completing actions with no interaction with the underlying state:

$$\begin{array}{lll} \textit{Informer}\ I & ::= & \mathsf{complete} \mid \mathsf{skip} \mid I \mathbin{\mathsf{and}} I \mid I \circ I \\ \textit{Action}\quad A & ::= & I \mid A \circ I \end{array}$$

The new composition combinator $(A \circ I)$ forwards incoming information to I and pipes the resulting information into A. (Semantically, the new construct is superfluous, as $(A \circ I)$ is equivalent to $I > A \setminus \mathsf{raise}$, but it is useful in the syntactic formulation of reduction, below.) We shall sometimes use the abbreviations

$$\begin{aligned} A\,I &\triangleq A \circ I \\ A(I_1, I_2) &\triangleq A \circ (I_1 \mathbin{\mathsf{and}} I_2) \end{aligned}$$

when A is is an action primitive, e.g., we shall write $\mathsf{raise}\, I$ rather than $\mathsf{raise} \circ I$.

The information flow follows the control flow in the basic actions from the previous section. Hence the information flow through $(A > A_1 \setminus A_2)$ is sequential—after A the information flow follows the flow of control into A_1 or A_2—and the information flow through $(A_1 \mathbin{\mathsf{and}} A_2)$ is parallel—upon normal completion of both branches the parallel information flows are merged (if one branch escapes the other branch is aborted and does not contribute to the information flow).

Structural congruence In order to specify the meaning of $A \circ I$, it is convenient to introduce a structural congruence relation, $\rightleftharpoons$, between action terms. It is the smallest congruence relation on action terms given by the following equational axioms. They describe the propagation of information into actions plus some further equational properties of informers.

(27) $\textsf{complete and } I = I \textsf{ and complete} = I\,;\quad (I \textsf{ and } I') \textsf{ and } I'' = I \textsf{ and } (I' \textsf{ and } I'')$

(28) $A \circ \textsf{skip} \rightleftharpoons A\,;\quad \textsf{skip} \circ I \rightleftharpoons I$

(29) $(A_1 \textsf{ or } A_2) \circ I \rightleftharpoons (A_1 \circ I) \textsf{ or } (A_2 \circ I)\,;\quad (A_1 \textsf{ and } A_2) \circ I \rightleftharpoons (A_1 \circ I) \textsf{ and } (A_2 \circ I)$

(30) $(\textsf{indivisibly } A) \circ I \rightleftharpoons \textsf{indivisibly } (A \circ I)\,;\quad \textsf{complete} \circ I \rightleftharpoons \textsf{complete}$

(31) $(A \circ I) \circ I' \rightleftharpoons A \circ (I \circ I')\,;\quad (A > A_1 \setminus A_2) \circ I \rightleftharpoons (A \circ I) > A_1 \setminus A_2$

Reduction semantics The following reduction rule specifies that reduction is in effect a relation between structural congruence equivalence classes of action terms:

(32) $\dfrac{A_1' \to A_2'}{A_1 \to A_2}$ if $A_1 \rightleftharpoons A_1'$ and $A_2' \rightleftharpoons A_2$

The terminated actions in the reduction rule (7) for the indivisibly combinator are now those generated by the grammar:

$$\textit{Terminated}\ T \ ::=\ I \mid \textsf{raise} \mid T \circ I$$

The next reduction rule exploits that reduction is defined on 'open' actions, viz. actions which refer to incoming information.

(33) $\dfrac{A \to A'}{A \circ I \to A' \circ I}$

Here are two simple reductions, derived from (32), (33), and (3):

$$I > A_1 \setminus A_2 \to A_1 \circ I\,;\quad \textsf{raise}\, I > A_1 \setminus A_2 \to A_2 \circ I$$

Equations

(34) $A = A'$, if $A \rightleftharpoons A'$

(35) $\textsf{complete and } A = A \textsf{ and complete} = A\,;\quad (\textsf{raise}\, I) \textsf{ and } A = A \textsf{ and } (\textsf{raise}\, I)$

(36) $(\textsf{raise}\, I) \textsf{ and } I' = \textsf{raise}\, I\,;\quad (\textsf{raise}\, I) \textsf{ and } (\textsf{raise}\, I') = (\textsf{raise}\, I) \textsf{ or } (\textsf{raise}\, I')$

4.1 The declarative and functional facets

The core syntax of the declarative facet is:

$$\begin{array}{lll} \textit{Action} & A & ::= \textsf{find}\, t \\ \textit{Informer} & I & ::= \textsf{bind}\, t \mid \textsf{rebind} \\ \textit{token} & t & ::= \ldots \end{array}$$

For simplicity, tokens are not first-class.

Computed bindings are represented syntactically by informers which merge together individual singleton bindings $\mathsf{bind}\, t\, I$ by the and combinator. The merging of the bindings of I and I' in $(I\,\mathsf{and}\,I')$ is obtained by overlaying the bindings of I' over those of I (i.e., and is actually interpreted like action notation's $\mathsf{moreover}$ combinator; this form of merging is better behaved than disjoint union because the latter is a partial operation). Some of action notation's sequential declarative combinators will be defined later on.

The functional facet is the interface between the algebra of actions and some algebraic specification of underlying data types. The facet also includes higher-order primitives (classified as the separate reflective facet in action notation).

$$\begin{array}{lll}
\textit{Action} & A & ::= (\textit{dataop}) \mid \mathsf{enact} \\
\textit{Informer} & I & ::= d \mid \mathsf{regive} \\
\textit{data} & d & ::= () \mid (d, d) \mid \mathsf{abstraction\ of}\ A \mid \ldots
\end{array}$$

The underlying data types should include the sort *data* with ($\mathsf{abstraction\ of}\ A$) as an element for all actions A, with the empty tuple () as element, and equipped with an associative tupling operator $(\cdot, \cdot)$ with () as unit. Furthermore, one should impose a uniformity requirement on data operations with respect to abstractions; see [Las97].

The metavariable *dataop* ranges over data operations provided by the data types, and d ranges over elements of sort *data*.

The data types can be specified in any formalism. In the following we assume that a unified algebra specification is used, as in [Mos92]. We use notation $\vdash d = d'$ and $\vdash d' : ds$ (where *ds* is any subsort of *data*) for unified algebra judgements concerning the algebraic specification.

Structural congruence The structural congruence axioms are grouped into declarative, functional, and hybrid axioms.

(37) $\mathsf{rebind} \circ \mathsf{rebind} \rightleftharpoons \mathsf{rebind}\,; \quad \mathsf{rebind} \circ \mathsf{complete} \rightleftharpoons \mathsf{complete}$

(38) $\mathsf{rebind} \circ (I\ \mathsf{and}\ I') \rightleftharpoons (\mathsf{rebind} \circ I)\ \mathsf{and}\ (\mathsf{rebind} \circ I')\,; \quad \mathsf{rebind} \circ \mathsf{bind}\, t \rightleftharpoons \mathsf{bind}\, t$

(39) $\mathsf{regive} \circ \mathsf{regive} \rightleftharpoons \mathsf{regive}\,; \quad \mathsf{regive} \circ \mathsf{complete} \rightleftharpoons \mathsf{complete}$

(40) $\mathsf{regive} \circ (I\ \mathsf{and}\ I') \rightleftharpoons (\mathsf{regive} \circ I)\ \mathsf{and}\ (\mathsf{regive} \circ I')\,; \quad \mathsf{regive} \circ d \rightleftharpoons d$

(41) $\mathsf{complete} \rightleftharpoons ()\,; \quad d\ \mathsf{and}\ d' \rightleftharpoons (d, d')\,; \quad d \circ I \rightleftharpoons d$

(42) $\mathsf{rebind} \circ \mathsf{regive} \rightleftharpoons \mathsf{regive} \circ \mathsf{rebind} \rightleftharpoons \mathsf{complete}\,; \quad \mathsf{rebind\ and\ regive} \rightleftharpoons \mathsf{copy}$

(43) $(\mathsf{rebind} \circ I)\ \mathsf{and}\ (\mathsf{regive} \circ I') \rightleftharpoons (\mathsf{regive} \circ I')\ \mathsf{and}\ (\mathsf{rebind} \circ I)$

The structural congruence axioms reduce the somewhat unwieldy syntactic sort of informers to a smaller subsort *info* generated by the grammar:

$$\textit{info}\ \ i\ ::= \mathsf{complete} \mid \mathsf{skip} \mid i\,\mathsf{and}\,i \mid \mathsf{rebind} \mid \mathsf{bind}\, t\, i \mid \mathsf{regive} \mid d$$

This result is useful as some definitions are more easily specified by structural induction on *info* than on *Informer*. For instance:

$$\cdot\ \mathit{at}\ \cdot : \mathit{info} \times \mathit{token} \rightharpoonup \mathit{info} \uplus \{\mathit{unbound}\}$$

$$(i_1\ \mathsf{and}\ i_2)\ \mathit{at}\ t \equiv \begin{cases} i & \text{if } i_2\ \mathit{at}\ t \equiv i \\ i & \text{if } i_1\ \mathit{at}\ t \equiv i \text{ and } i_2\ \mathit{at}\ t \equiv \mathit{unbound} \\ \mathit{unbound} & \text{if } i_1\ \mathit{at}\ t \equiv i_2\ \mathit{at}\ t \equiv \mathit{unbound} \end{cases}$$

$$(\mathsf{bind}\, t'\ i)\ \mathit{at}\ t \equiv \begin{cases} i & \text{if } t = t' \\ \mathit{unbound} & \text{if } t \neq t' \end{cases}$$

$$i\ \mathit{at}\ t \equiv \mathit{unbound}, \quad \text{if } \mathsf{rebind} \circ i \rightleftharpoons \mathsf{complete}$$

$$i\ \mathit{at}\ t \text{ is undefined otherwise}$$

(The penultimate clause abbreviates three clauses for complete, regive, and d.)

Reduction semantics The reduction rules for the find t primitive are:

(44) $\mathsf{find}\, t\, I \to I', \quad \text{if } I\ \mathit{at}\ t \equiv I'$

(45) $\mathsf{find}\, t\, I \to \mathsf{raise}\, I, \quad \text{if } I\ \mathit{at}\ t \equiv \mathit{unbound}$

The general schema for error exceptions is that any incoming information is passed through to the exception handler.

The functional primitives (dataop) and enact only reduce if the incoming information is a data constant d; and in the case of enact, d must even be equal to an abstraction. (It may be preferable to let enact and data operations ignore bindings rather than, as here, insist that they are empty—then regive is also definable as (id) where id is the identity function on data.)

For conciseness and to eschew possible error situations when unified algebra data operations return proper sorts, we let (dataop) choose any individual in the resulting sort if dataop does not map the incoming data to a vacuous sort; an error exception is raised if the resulting sort is nothing

(46) $(\mathit{dataop})\, d \to d', \quad \text{if } \vdash d' : \mathit{dataop}(d)$

(47) $(\mathit{dataop})\, d \to \mathsf{raise}\, d, \quad \text{if } \vdash \mathit{dataop}(d) = \mathsf{nothing}$

(48) $\mathsf{enact}\, d \to A \circ \mathsf{complete}, \quad \text{if } \vdash d = \mathsf{abstraction\ of}\ A$

Equations

(49) $(\mathsf{bind}\, t\, I)\ \mathsf{and}\ (\mathsf{bind}\, t\, I') = \mathsf{bind}\, t\, I'$

(50) $(\mathsf{bind}\, t\, I)\ \mathsf{and}\ (\mathsf{bind}\, t'\, I') = (\mathsf{bind}\, t'\, I')\ \mathsf{and}\ (\mathsf{bind}\, t\, I), \quad \text{if } t \neq t'$

(51) $\mathsf{find}\, t\, I = \mathsf{raise}\, I, \quad \text{if } \mathsf{rebind} \circ I = \mathsf{complete}$

(52) $\mathsf{find}\, t\, (I, \mathsf{bind}\, t'\, I') = \begin{cases} I' & \text{if } t = t' \\ I'' & \text{if } t \neq t' \text{ and } \mathsf{find}\, t\, I = I'' \\ \mathsf{raise}\,(I, \mathsf{bind}\, t'\, I') & \text{if } t \neq t' \text{ and } \mathsf{find}\, t\, I = \mathsf{raise}\, I \end{cases}$

(53) $d = d', \quad \text{if } \vdash d = d'$

(54) $(\mathit{dataop})\, d \sqsubseteq d', \quad \text{if } \vdash d' : \mathit{dataop}(d)$

(55) $(\mathit{dataop})\, d = \mathsf{raise}\, I, \quad \text{if } \vdash \mathit{dataop}(d) = \mathsf{nothing}$

(56) $\mathsf{enact}\ \ \mathsf{abstraction\ of}\ A = A \circ \mathsf{complete}$

4.2 Copying

So far the algebra lacks much of the power of action notation for expressing information flow. For instance, even though we have both the and combinator and the ternary sequencing combinator, there does not seem to be an easy way of expressing the control and information flow represented by action notation's $\mathsf{and\ then}$ combinator. In fact, of the many sequential standard action combinators only the thence combinator is easily expressed by the core constructs introduced thus far:

$$A_1 \;\mathsf{thence}\; A_2 \triangleq A_1 \mathbin{\mathsf{>}} A_2 \setminus \mathsf{raise}$$

Another significant shortcoming is that it is not possible to express the closure of an abstraction with the current bindings.

These deficiencies are all rectified by the following new $\mathsf{copying}$ and $\mathsf{copy}\#p$ constructs, syntactically similar to $\mathsf{unfolding}$ and $\mathsf{unfold}\#p$.

$$\begin{array}{lll} \textit{Action} \quad A & ::= & \mathsf{copying}\, A \\ \textit{Informer} \quad I & ::= & \mathsf{copying}\, I \mid \mathsf{copy}\#p \qquad (p \geq 1) \end{array}$$

The $\mathsf{copy}\#p$ primitive refers to the information that flows into the p'th enclosing $\mathsf{copying}$ combinator, cf. the structural congruence laws:

(57) $\mathsf{copy}\#p \circ I \rightleftharpoons \mathsf{copy}\#p\,; \quad \mathsf{copying\ copy} \rightleftharpoons \mathsf{skip}$

(58) $I \rightleftharpoons \mathsf{copying}\,(I{\uparrow}_{\mathsf{copy}})\,; \quad \mathsf{copying\ copying}\, I \rightleftharpoons \mathsf{copying}\,(I{\downarrow}_{\mathsf{copy}})$

(59) $(\mathsf{copying}\, A) \circ (\mathsf{copying}\, I) \rightleftharpoons \mathsf{copying}\,(A\{\mathsf{copy} := I \circ \mathsf{copy}\} \circ I)$

(60) $\mathsf{copying}\,(I_1 \;\mathsf{and}\; I_2) \rightleftharpoons (\mathsf{copying}\, I_1) \;\mathsf{and}\; (\mathsf{copying}\, I_2)$

Remark 2. Now, informers are reducible to info's of the extending grammar:

$$\begin{array}{lll} \textit{info} \;\; i & ::= & \mathsf{complete} \mid \mathsf{skip} \mid i \;\mathsf{and}\; i \mid \mathsf{rebind} \mid \mathsf{bind}\; t\, i \mid \mathsf{regive} \mid d \\ & & \mid \mathsf{copy}\#p \mid \mathsf{rebind\ copy}\#p \mid \mathsf{regive\ copy}\#p \mid \mathsf{copying}\; d \end{array}$$

which is still a convenient characterisation, e.g., usable in the definition of *at*.

With the new constructs, we define:

$$\begin{aligned} A_1 \;\mathsf{and\ then}\; A_2 &\triangleq \mathsf{copying}\,(A_1 \;\mathsf{thence}\; (\mathsf{skip\ and}\; (A_2 \circ \mathsf{copy}))) \\ A_1 \;\mathsf{hence}\; A_2 &\triangleq \mathsf{copying}\,(A_1 \mathbin{\mathsf{>}} A_2 \circ (\mathsf{rebind\ and\ regive\ copy}) \setminus \mathsf{raise}) \\ A_1 \;\mathsf{then}\; A_2 &\triangleq \mathsf{copying}\,(A_1 \mathbin{\mathsf{>}} A_2 \circ (\mathsf{regive\ and\ rebind\ copy}) \setminus \mathsf{raise}) \\ \mathsf{escape} &\triangleq \mathsf{raise} \circ \mathsf{regive} \\ A_1 \;\mathsf{trap}\; A_2 &\triangleq \mathsf{copying}\,(A_1 \mathbin{\mathsf{>}} \mathsf{skip} \setminus A_2 \circ (\mathsf{regive\ and\ rebind\ copy})) \end{aligned}$$

If as in action notation we overload then and hence to also be data operations on abstractions, we can define:

$$\begin{aligned} &\mathsf{close} \triangleq (\mathsf{hence})\,(\mathsf{copying\ abstraction\ of\ rebind\ copy, regive}) \\ &\mathsf{apply} \triangleq (((\mathsf{rest})\;\mathsf{thence\ copying\ abstraction\ of\ copy})\;\mathsf{and}\;(\mathsf{first}))\;\mathsf{then}\;(\mathsf{then}) \end{aligned}$$

where first and rest are the standard data notation operations on tuples. Given bindings and an abstraction as incoming information close builds a closure abstraction as in action notation. The action apply takes a data tuple with an abstraction as first component and builds an abstraction which is the application of the abstraction to the rest of the data tuple as in action notation.

Reduction semantics Reduction is permitted under the copying combinator:

(61) $$\frac{A \to A'}{\mathsf{copying}\, A \to \mathsf{copying}\, A'}$$

For instance, the next two derived reductions are derived from (61) and (4):

$$(\mathsf{raise}\, I)\ \mathsf{and}\ A \to \mathsf{raise}\, I\,; \quad A\ \mathsf{and}\ (\mathsf{raise}\, I) \to \mathsf{raise}\, I$$

Equations We have the following equational laws for copying.

(62) $\mathsf{copying}\,(A{\uparrow_{\mathsf{copy}}}) = A\,; \quad \mathsf{copying}\,\mathsf{copying}\,A = \mathsf{copying}\,(A{\downarrow_{\mathsf{copy}}})$

(63) $\mathsf{copying}\,(A_1\ \mathsf{or}\ A_2) = (\mathsf{copying}\,A_1)\ \mathsf{or}\ (\mathsf{copying}\,A_2)$

(64) $(\mathsf{copying}\,A) > A_1 \setminus A_2 = \mathsf{copying}\,(A > A_1{\uparrow_{\mathsf{copy}}} \setminus A_2{\uparrow_{\mathsf{copy}}})$

(65) $\mathsf{copying}\,(A_1\ \mathsf{and}\ A_2) = (\mathsf{copying}\,A_1)\ \mathsf{and}\ (\mathsf{copying}\,A_2)$

A refinement relationship between the combinators 'and' and 'and then' is asserted by the inequational law:

(66) $A_1\ \mathsf{and}\ A_2 \sqsubseteq A_1\ \mathsf{and\ then}\ A_2$

5 State and the imperative facet

The imperative and communicate facets are state-based. Both create and process objects that are accessed and referred to by means of references. The core algebra provides the following common structure for the state facets:

$$\begin{aligned} &\mathit{Action}\ \ A \ ::= \ A\ \mathsf{with\ new}\ \vec{O} \ \mid\ \mathsf{compare} \\ &\mathit{Object}\ \ O \ ::= \ \mathsf{\mathit{obj}}\,\Phi \end{aligned}$$

Objects O are of the form $\mathit{obj}\,\Phi$ where obj is the object's kind (cell, agent, or message) and Φ is a syntactic phrase or is empty. The A with new $\vec{O}$ action creates the possibly mutually recursive objects $\vec{O}$ before performing A.

The information flow into A with new $\mathit{obj}_1\Phi_1, \ldots, \mathit{obj}_n\Phi_n$ is specified by:

(67) $(A\ \mathsf{with\ new}\ \mathit{obj}_1\Phi_1, \ldots) \circ I \rightleftharpoons (A \circ I)\ \mathsf{with\ new}\ \mathit{obj}_1(\Phi_1 \circ I), \ldots$

Some kinds of objects may be referenced, and for each such object kind obj there are object references $\mathit{obj}\#p$ of sort *data*.

$$\mathit{data}\ \ d\ ::=\ \mathit{obj}\#p$$

Within A with new $\vec{O}$ the p'th object of some kind, obj, can be referred to from A and from the objects in $\vec{O}$ via the reference $\mathit{obj}\#p$, but it is hidden from the outside. The compare primitive compares two object references for equality.

Imperative Each state facet specifies a collection of objects that can go into the state together with action primitives for interacting with the objects. The imperative facet introduces the object kind cell, and provides primitives update and access for assigning to and dereferencing cells, respectively. There is no construct for explicitly deallocating cells.

$$\begin{array}{lll} \textit{Action} & A & ::= \mathsf{update} \mid \mathsf{access} \\ \textit{Object} & O & ::= \mathsf{cell}\, I \\ \textit{data} & d & ::= \mathsf{cell}\#p \end{array}$$

5.1 Reduction semantics

First, we specify the 'generic' reductions for state, those that make no assumptions about the actual objects in the state.

Let $\Phi\!\uparrow_{\vec{O}} \triangleq \Phi\!\uparrow_{\mathit{obj}_1} \cdots \uparrow_{\mathit{obj}_n}$ if $\vec{O} \equiv \mathit{obj}_1\, \Phi_1, \ldots, \mathit{obj}_n\, \Phi_n$.

(68) $(A\ \mathsf{with\ new}\ \vec{O})\ \mathsf{>}\ A_1 \setminus A_2 \to (A\ \mathsf{>}\ A_1\!\uparrow_{\vec{O}} \setminus A_2\!\uparrow_{\vec{O}})\ \mathsf{with\ new}\ \vec{O}$

(69) $(A_1\ \mathsf{with\ new}\ \vec{O})\ \mathsf{and}\ A_2 \to (A_1\ \mathsf{and}\ A_2\!\uparrow_{\vec{O}})\ \mathsf{with\ new}\ \vec{O}$

(70) $A_1\ \mathsf{and}\ (A_2\ \mathsf{with\ new}\ \vec{O}) \to (A_1\!\uparrow_{\vec{O}}\ \mathsf{and}\ A_2)\ \mathsf{with\ new}\ \vec{O}$

(71) $(A\ \mathsf{with\ new}\ \vec{O}_1)\ \mathsf{with\ new}\ \vec{O}_2 \to A\ \mathsf{with\ new}\ \vec{O}_1, \vec{O}_2\!\uparrow_{\vec{O}_1}$

(72) $\mathsf{compare}(\mathit{obj}\#p, \mathit{obj}\#p) \to \mathsf{complete}$

(73) $\mathsf{compare}\, I \to \mathsf{raise}\, I,\quad \text{if } I \equiv (\mathit{obj}\#p, \mathit{obj}\#q) \text{ and } p \neq q$

(74) $$\frac{A \to A'}{A\ \mathsf{with\ new}\ \vec{O} \to A'\ \mathsf{with\ new}\ \vec{O}}$$

The next axioms and rule define the behaviour of the imperative facet and extend the semantics of the $\mathsf{indivisibly}$ combinator to imperative state.

Notation $|\vec{O}|_{\mathit{obj}}$ denotes the number of objects of kind obj in $\vec{O}$.

(75) $E[\mathsf{access}(\mathsf{cell}\#|\vec{O}_1|_{\mathsf{cell}}{+}1)]\ \mathsf{with\ new}\ \vec{O}_1, \mathsf{cell}\, I, \vec{O}_2 \to$
$E[I]\ \mathsf{with\ new}\ \vec{O}_1, \mathsf{cell}\, I, \vec{O}_2$

(76) $E[\mathsf{update}(\mathsf{cell}\#|\vec{O}_1|_{\mathsf{cell}}{+}1, I')]\ \mathsf{with\ new}\ \vec{O}_1, \mathsf{cell}\, I, \vec{O}_2 \to$
$E[\mathsf{complete}]\ \mathsf{with\ new}\ \vec{O}_1, \mathsf{cell}\, I', \vec{O}_2$

(77) $$\frac{A\ \mathsf{with\ new}\ \vec{C} \to^{*} T\ \mathsf{with\ new}\ \vec{O}, \vec{C}'}{E[\mathsf{indivisibly}\, A]\ \mathsf{with\ new}\ \vec{C} \to E\!\uparrow_{\vec{O}}[T]\ \mathsf{with\ new}\ \vec{O}, \vec{C}'}\quad \text{if } |\vec{C}| = |\vec{C}'|$$

5.2 Equations

(78) $A\ \mathsf{with\ new}\ () = A$

(79) $(A\ \mathsf{or}\ A')\ \mathsf{with\ new}\ \vec{O} = (A\ \mathsf{with\ new}\ \vec{O})\ \mathsf{or}\ (A'\ \mathsf{with\ new}\ \vec{O})$

(80) $(A\ \mathsf{with\ new}\ \vec{O})\ \mathsf{>}\ A_1 \setminus A_2 = (A\ \mathsf{>}\ A_1\!\uparrow_{\vec{O}} \setminus A_2\!\uparrow_{\vec{O}})\ \mathsf{with\ new}\ \vec{O}$

(81) $\textsf{indivisibly}\,(T\ \textsf{with new}\ \vec{O}) = T\ \textsf{with new}\ \vec{O}$

(82) $\textsf{indivisibly}\,((A\ \textsf{with new}\ \vec{O})\ \textsf{with new}\ \vec{O}') = \textsf{indivisibly}\,(A\ \textsf{with new}\ \vec{O}, \vec{O}'{\uparrow_{\vec{O}}})$

(83) $(\textsf{copying}\ A)\ \textsf{with new}\ \vec{O} = \textsf{copying}\ (A\ \textsf{with new}\ \vec{O}{\uparrow_{\textsf{copy}}})$

The order of the objects $\vec{O}$ in $A\ \textsf{with new}\ \vec{O}$ is indifferent modulo suitable renaming of all references to $\vec{O}$ in A and in $\vec{O}$. In order to formulate this law, we define a relation $\pi : \vec{O} \hookrightarrow \vec{O}'$ where π is an object reference substitution which permutes object references, and $\vec{O}'$ is a corresponding permutation of $\vec{O}\pi$ (i.e., $\vec{O}'$ is a permutation of $\vec{O}$ with all mutual object references suitably updated). The law reads as follows:

(84) $A\ \textsf{with new}\ \vec{O} = A\pi\ \textsf{with new}\ \vec{O}', \quad \text{if } \pi : \vec{O} \hookrightarrow \vec{O}'$

The formal definition of $\pi : \vec{O} \hookrightarrow \vec{O}'$ is specified inductively by the following axioms and rule:

$$id : \vec{O}_1, \textsf{obj}\,\Phi, \textsf{obj}'\,\Phi', \vec{O}_2 \hookrightarrow \vec{O}_1, \textsf{obj}'\,\Phi', \textsf{obj}\,\Phi, \vec{O}_2\,, \quad \text{if } \textsf{obj} \neq \textsf{obj}'$$

$$\pi : \vec{O}_1, \textsf{obj}\,\Phi, \textsf{obj}\,\Phi', \vec{O}_2 \hookrightarrow \vec{O}_1\pi, \textsf{obj}\,\Phi', \textsf{obj}\,\Phi, \vec{O}_2\pi\,,$$
$$\text{if } |\vec{O}_1|_{\textsf{obj}} = p-1 \text{ and } \pi = \{\textsf{obj}\#p := \textsf{obj}\#p{+}1, \textsf{obj}\#p{+}1 := \textsf{obj}\#p\}$$

$$id : \vec{O} \hookrightarrow \vec{O}\,; \qquad \frac{\pi : \vec{O} \hookrightarrow \vec{O}' \quad \pi' : \vec{O}' \hookrightarrow \vec{O}''}{\pi\,;\pi' : \vec{O} \hookrightarrow \vec{O}''}$$

where id is the identity substitution and $\pi\,;\pi'$ is the composition of substitutions π and π' satisfying $\Phi(\pi\,;\pi') = (\Phi\pi)\pi'$.

In the following laws for the imperative facet C ranges over cell objects.

(85) $\textsf{indivisibly update} = \textsf{update}\,; \quad \textsf{indivisibly access} = \textsf{access}$

(86) $A > (A_1\ \textsf{with new}\ \vec{C}) \setminus A_2 = (A{\uparrow_{\vec{C}}} > A_1 \setminus A_2{\uparrow_{\vec{C}}})\ \textsf{with new}\ \vec{C}$

(87) $A > A_1 \setminus (A_2\ \textsf{with new}\ \vec{C}) = (A{\uparrow_{\vec{C}}} > A_1{\uparrow_{\vec{C}}} \setminus A_2)\ \textsf{with new}\ \vec{C}$

(88) $(A_1\ \textsf{with new}\ \vec{C})\ \textsf{and}\ A_2 = (A_1\ \textsf{and}\ A_2{\uparrow_{\vec{C}}})\ \textsf{with new}\ \vec{C}$

(89) $A_1\ \textsf{and}\ (A_2\ \textsf{with new}\ \vec{C}) = (A_1{\uparrow_{\vec{C}}}\ \textsf{and}\ A_2)\ \textsf{with new}\ \vec{C}$

(90) $\textsf{indivisibly}\,(A\ \textsf{with new}\ \vec{C}) = (\textsf{indivisibly}\ A)\ \textsf{with new}\ \vec{C}$

(91) $(A\ \textsf{with new}\ \vec{C})\ \textsf{with new}\ \vec{O} = A\ \textsf{with new}\ \vec{C}, \vec{O}{\uparrow_{\vec{C}}}$

(92) $(A\ \textsf{with new}\ \vec{O})\ \textsf{with new}\ \vec{C} = A\ \textsf{with new}\ \vec{O}, \vec{C}{\uparrow_{\vec{O}}}$

(93) $A{\uparrow_{\vec{C}}}\ \textsf{with new}\ \vec{C} = A$

(94) $(\textsf{update}(\textsf{cell}\#1, I') > A \setminus A')\ \textsf{with new cell}\ I, \vec{C} =$
$\qquad (A \circ \textsf{complete})\ \textsf{with new cell}\ I', \vec{C}$

(95) $(\textsf{access}(\textsf{cell}\#1) > A \setminus A')\ \textsf{with new cell}\ I, \vec{C} = (A \circ I)\ \textsf{with new cell}\ I, \vec{C}$

(96) $(\textsf{indivisibly}\,(\textsf{update}(\textsf{cell}\#1, I') > A \setminus A') > A_1 \setminus A_2)\ \textsf{with new cell}\ I, \vec{C} =$
$\qquad (\textsf{indivisibly}\,(A \circ \textsf{complete}) > A_1 \setminus A_2)\ \textsf{with new cell}\ I', \vec{C}$

(97) $(\textsf{indivisibly}\,(\textsf{access}(\textsf{cell}\#1) > A \setminus A') > A_1 \setminus A_2)\ \textsf{with new cell}\ I, \vec{C} =$
$\qquad (\textsf{indivisibly}\,(A \circ I) > A_1 \setminus A_2)\ \textsf{with new cell}\ I, \vec{C}$

A final inequational law:

(98)† $\quad A \sqsubseteq \mathsf{indivisibly}\, A$

holds only in the absence of concurrent agents with fairness constraints.

6 The communicative facet

The communicative facet introduces asynchronous processes, called agents, and messages. Agents are actions with a process identity. Each message must be addressed to a specific agent.

$$\begin{array}{lll} \textit{Action} & A & ::= \ \mathsf{receive} \\ \textit{Object} & O & ::= \ \mathsf{agent}\, A \ \mid \ \mathsf{message}\, I \\ \textit{data} & d & ::= \ \mathsf{agent}\#p \ \mid \ \mathsf{self}\#p \end{array}$$

The A with new $\vec{O}$ construction sends all messages in $\vec{O}$ and spawns all agents in $\vec{O}$ before performing A. The fairness constraints of action notation on agents' progress and on message delivery are intended but they are not modelled in the reduction semantics that is given below. This aspect is left for future work, cf. §8.

There are no message references because messages are removed from the state upon receipt, and they should not leave behind dangling references. The self reference is not an object reference like the others, although it refers to agents. Its index counts the lexically enclosing agents and abstractions: if it points to an agent it refers to that agent; if it points to an abstraction it refers to the agent which is self$\#1$ at the point where that abstraction was enacted. Thus, the self$\#1$ reference effectively has dynamic scope and corresponds to the yielder performing-agent in action notation. It is useful to be able to refer to enclosing agents when setting up communication protocols in connection with creation of agents (although there is not an exact correspondence, self$\#p$ references should be able to replace the uses of the contracting-agent yielder in action notation). It is not obviously useful to refer to the self$\#1$'s of enclosing abstractions, but this feature is necessary in order to obtain the desired dynamic scope of self$\#1$.

Cells and agents may cross-reference each other and it is most natural to let cells be shared between agents.

The receive primitive either removes from the current state a message addressed to the performing agent and completes with the message contents as outcome, or escapes. The intended fairness constraint on message delivery is that, whenever a message is sent to an agent that infinitely often attempts to receive, the message is eventually received and removed by the agent, i.e., the agent must not indefinitely choose to receive other messages or escape.

Remark 3. In action notation the action:

receive message [containing ds]

awaits the arrival of a message with contents of sort ds. In our algebra this behaviour is expressed by the compound action:

```
unfolding (receive > (ds&) > skip
                                  \ (unfold with new message (self#1, skip))
                      \ unfold)
```

where ds& is the data operation that intersects its argument with the sort ds.

6.1 Reduction semantics

The reduction rule (77) for the indivisibly combinator in the imperative facet forces the constraint on reductions between actions of the form A with new $\vec{O}$ that the new objects $\vec{O}$ are not reordered by reductions, as otherwise any references in E to the new objects become corrupted. This constraint is difficult to maintain in the reductions for communicative actions. Instead, we introduce an auxiliary labelled transition relation $A, \vec{O} \xrightarrow{\rho} A', \vec{O}'$ between configurations $A, \vec{O}$ and $A', \vec{O}'$ each consisting of an action paired with a sequence of objects. The label ρ is a renaming of references to the objects $\vec{O}$ on the left hand side.

A configuration $A, \vec{O}$ corresponds to the action A with new $\vec{O}$ except that references to the objects $\vec{O}$ are bound in the action A with new $\vec{O}$ but not in the configuration $A, \vec{O}$.

Labelled transitions and reductions are interrelated by the two rules:

$$(99) \qquad \frac{A \to A'}{A \xrightarrow{id} A'}$$

$$(100) \qquad \frac{A, \vec{O} \xrightarrow{\rho} A', \vec{O}'}{A \text{ with new } \vec{O} \to A' \text{ with new } \vec{O}'}$$

The renaming ρ of references to the objects of the configuration $A, \vec{O}$ is only used internally in the definition of the transition relation; it is discarded in the reduction where the relevant object references are bound.

The transition relation is free to reorder objects and we also let it operate under evaluation contexts and in the absence of selected objects:

$$(101) \qquad \frac{A, \vec{O}_1 \xrightarrow{\rho} A', \vec{O}_1'}{E[A], \vec{O} \xrightarrow{\pi;\rho} (E\pi\rho)[A'], \vec{O}_1', \vec{O}_2\rho} \quad \text{if } \pi : \vec{O} \hookrightarrow \vec{O}_1, \vec{O}_2$$

The rules (99)–(101) subsume the reduction rule (74).

Now the reduction rule (77) for indivisibly is no longer valid. Instead, we define

$$(102) \qquad \frac{A, \vec{Q} \equiv A_0, \vec{O}_0 \xrightarrow{\rho_1} \cdots \xrightarrow{\rho_n} A_n, \vec{O}_n \equiv T, \vec{O}}{\text{indivisibly } A, \vec{Q} \xrightarrow{id;\rho_1;\ldots;\rho_n} T, \vec{O}} \quad (n \geq 0)$$

where Q ranges over non-agents, i.e., every Q is either a cell or a message, so that the reduction of A is not interleaved with the reduction of concurrent agents.

The reduction rule (77) is subsumed by the transition rule (102) together with (100) and (101).

Similarly, the following transition axioms subsume the reduction axioms (68)–(71) and (75)–(76).

(103) $A\ \mathsf{with\ new}\ \vec{O} \xrightarrow{\uparrow_{\vec{O}}} A, \vec{O}$

(104) $\mathsf{access}(\mathsf{cell}\#p), \mathsf{cell}\ I \xrightarrow{id} I, \mathsf{cell}\ I$

(105) $\mathsf{update}(\mathsf{cell}\#p, I'), \mathsf{cell}\ I \xrightarrow{id} \mathsf{complete}, \mathsf{cell}\ I'$

There are the following transition axioms and rules for message receipt and concurrency.

(106) $\mathsf{receive}\ I, \mathsf{message}(\mathsf{self}\#1, I') \xrightarrow{id} I'$

(107) $$\frac{(A_1, \vec{O}\uparrow_{\mathsf{self}})\lceil \mathsf{agent} := \mathsf{self}\#1\rceil \xrightarrow{\rho} A_1', \vec{O}'}{A, \mathsf{agent}\ A_1, \vec{O} \xrightarrow{\rho'} A\rho', \mathsf{agent}\ A_1', \vec{O}'\uparrow_{obj}\lceil \mathsf{self} := \mathsf{agent}\#1\rceil} \quad \text{if } \rho' = \rho\,;\uparrow_{obj}$$

The following reduction and transition axioms specify the effect of comparing self references with other self references or agent references.

(108) $\mathsf{compare}(\mathsf{self}\#p, \mathsf{self}\#p) \to \mathsf{complete}$

(109) $\mathsf{compare}\ I, \mathsf{agent}\ A \xrightarrow{id} \mathsf{raise}\ I, \mathsf{agent}\ A$,
if $I \equiv (\mathsf{self}\#p, \mathsf{agent}\#1)$ or $I \equiv (\mathsf{agent}\#1, \mathsf{self}\#p)$

Remark 4. In some cases the same agent is referred to by a self reference and an agent reference or a different self reference, e.g., in the actions

$A\ \mathsf{with\ new\ agent\ compare}\ (\mathsf{self}\#1, \mathsf{agent}\#1)$

$\mathsf{enact} \circ \mathsf{abstraction\ of}\ (\mathsf{compare}(\mathsf{self}\#1, \mathsf{self}\#2))$

It is not clear whether the present reduction rules for comparing references to agents are an adequate operational semantics for compare.

Finally, we need to modify (48) to take into account that the abstraction of constructor binds the self reference to the value of self (the performing agent) at the point where it is enacted:

(110) $\mathsf{enact}\ d \to A\downarrow_{\mathsf{self}} \circ \mathsf{complete}$, if $\vdash d = \mathsf{abstraction\ of}\ A$

6.2 Equations

(111) $\mathsf{enact\ abstraction\ of}\ A = A\downarrow_{\mathsf{self}} \circ \mathsf{complete}$

(112) $\mathsf{indivisibly\ receive} = \mathsf{receive}$

(113) $A\ \mathsf{with\ new\ agent}(A_1\ \mathsf{or}\ A_1'), \vec{O} =$
$(A\ \mathsf{with\ new\ agent}\ A_1, \vec{O})\ \mathsf{or}\ (A\ \mathsf{with\ new\ agent}\ A_1', \vec{O})$

(114) $A\ \mathsf{with\ new\ agent}(\mathsf{copying}\ A_1), \vec{O} = \mathsf{copying}\ (A\uparrow_{\mathsf{copy}}\ \mathsf{with\ new\ agent}\ A_1, \vec{O}\uparrow_{\mathsf{copy}})$

(115) $A\ \mathsf{with\ new\ agent}\ (A'\ \mathsf{with\ new}\ \vec{O}'), \vec{O} =$
$A\uparrow_{\vec{O}'}\ \mathsf{with\ new}\ \vec{O}'\lceil \mathsf{self} := \mathsf{agent}\#|\vec{O}'|_{\mathsf{agent}}{+}1\rceil, \mathsf{agent}\ A', \vec{O}\uparrow_{\vec{O}'}$

(116) $A{\uparrow}_{\mathsf{obj}}$ with new agent (receive I > $A_1 \setminus A_2$), message(agent#1, I') =
($A{\uparrow}_{\mathsf{obj}}$ with new agent ($A_1 \circ I'$)) or
($A{\uparrow}_{\mathsf{obj}}$ with new agent ($A_2 \circ I$), message(agent#1, I'))

It would be nice if (82) held without the indivisibly wrapper:

(117)[†] (A with new $\vec{O}$) with new $\vec{O}'$ = A with new $\vec{O}, \vec{O}'{\uparrow}_{\vec{O}}$

This would express an important property of asynchrony: the indifference of mutual timing of message transmissions and process spawning. But the combination of interleaving and exceptions from the basic facet renders (117)[†] invalid: if interleaved with an escaping action, the left hand side may be aborted between the creation of $\vec{O}'$ and $\vec{O}$ and this may be observable if there are messages or agents in both $\vec{O}$ and $\vec{O}'$.

7 Interleaving and expansion

One of the characteristic features of action notation, going back to the abstract semantic algebras of [Mos83], is the and combinator which interleaves the control flow of its arguments. But interleaving is left out in most work on action notation, and there does not seem to be much research on interleaving in higher-order, imperative languages in the literature. Interleaving is described straightforwardly in our reduction semantics and in other small-step operational semantics of action notation [Mos92,Mos99], yet the theory of interleaving is rather intricate.

One of its undesirable effects on the equational laws was mentioned in connection with the equational theory of the communicative facet.

Another problematic feature of interleaving is that it does not interact well with first-class continuations, as discussed in [Las94]. Indeed, I believe that first-class continuations can be added cleanly to the present core algebra along with a decent set of algebraic laws, provided that the interleaving operation of the and combinator is left out.

On the other hand, the core algebra may offer a positive result about the equational theory of interleaving: I conjecture that the algebra together with the set of expansion laws given below can reduce interleaving to non-determinism for first-order, non-recursive actions. This would be a kind of relative completeness result for the equational laws, as it were, indicating that they exhaust the equational properties specific to interleaving. The expansion laws rewrite actions into "disjunctive normal forms", that is, non-deterministic choices $N \equiv$ (B_1 or $\cdots$ or B_p) between actions B_1 through B_p generated by the grammar:

$$\begin{aligned} B &::= I \mid \mathsf{raise}\,I \mid \mathsf{copying}\,C \\ C &::= (\mathsf{indivisibly}\,A) > B \setminus B \mid B\ \mathsf{with\ new}\ \vec{O} \end{aligned}$$

The rewriting strategy is bottom-up. Consider (N and N') where $N \equiv$ (B_1 or $\cdots$ or B_p) and $N' \equiv$ (B'_1 or $\cdots$ or B'_q) with $p, q \geq 1$. Since or distributes over and, it suffices to rewrite (B_i and B'_j) into the desired form for $1 \leq i \leq p$ and $1 \leq j \leq q$. We consider four cases; the remaining cases are similar or simpler.

Case $B_i \equiv I_i$ *and* $B'_j \equiv \mathsf{copying}\,((\mathsf{indivisibly}\,A'_j) > B'_{j1} \setminus B'_{j2})$

(118) $B_i\ \mathsf{and}\ B'_j = \mathsf{copying}((\mathsf{indivisibly}\,A'_j) > ((I_i{\uparrow}_{\mathsf{copy}} \circ \mathsf{copy})\ \mathsf{and}\ B'_{j1})$
$\setminus ((I_i{\uparrow}_{\mathsf{copy}} \circ \mathsf{copy})\ \mathsf{and}\ B'_{j2}))$

Case $B_i \equiv \mathsf{raise}\,I_i$ *and* $B'_j \equiv \mathsf{copying}\,((\mathsf{indivisibly}\,A'_j) > B'_{j1} \setminus B'_{j2})$

(119) $B_i\ \mathsf{and}\ B'_j = \mathsf{raise}\,I_i\ \mathsf{or}$
$\mathsf{copying}((\mathsf{indivisibly}\,A'_j) > ((\mathsf{raise}\,I_i{\uparrow}_{\mathsf{copy}} \circ \mathsf{copy})\ \mathsf{and}\ B'_{j1})$
$\setminus ((\mathsf{raise}\,I_i{\uparrow}_{\mathsf{copy}} \circ \mathsf{copy})\ \mathsf{and}\ B'_{j2}))$

Case $B_i \equiv \mathsf{copying}\,C$ *and* $B'_j \equiv \mathsf{copying}\,C'$ *where*
$C \equiv (\mathsf{indivisibly}\,A_i) > B_{i1} \setminus B_{i2}$ *and* $C' \equiv (\mathsf{indivisibly}\,A'_j) > B'_{j1} \setminus B'_{j2}$

(120) $B_i\ \mathsf{and}\ B'_j =$
$\mathsf{copying}\,((\mathsf{indivisibly}\,A_i) > (B_{i1}\ \mathsf{and}\ (C' \circ \mathsf{copy})) \setminus (B_{i2}\ \mathsf{and}\ (C' \circ \mathsf{copy})))\ \mathsf{or}$
$\mathsf{copying}\,((\mathsf{indivisibly}\,A'_j) > ((C \circ \mathsf{copy})\ \mathsf{and}\ B'_{j1}) \setminus ((C \circ \mathsf{copy})\ \mathsf{and}\ B'_{j2}))$

Case $B_i \equiv \mathsf{copying}\,C$ *and* $B'_j \equiv \mathsf{copying}\,C'$ *where*
$C \equiv (\mathsf{indivisibly}\,A_i) > B_{i1} \setminus B_{i2}$ *and* $C' \equiv B'\ \mathsf{with\ new}\ \vec{O}'$

(121) $B_i\ \mathsf{and}\ B'_j =$
$\mathsf{copying}\,((\mathsf{indivisibly}\,A_i) > (B_{i1}\ \mathsf{and}\ (C' \circ \mathsf{copy})) \setminus (B_{i2}\ \mathsf{and}\ (C' \circ \mathsf{copy})))\ \mathsf{or}$
$\mathsf{copying}\,((C\ \mathsf{and}\ B')\ \mathsf{with\ new}\ \vec{O}')$

Applying the laws left-to-right, they may be applied recursively to the residual occurrences of and on the right-hand sides of the equations (up to structural congruence the arguments of and are all of the form dictated by the B grammar, above).

8 Assessment

The notation is very tentative. We have seen that it is well suited for expressing reduction and equations between actions but its utility for expressing the actions that are needed in action semantics has not been tested. Certain data calculations that are easily expressed with action notation's yielders become rather tedious in the core algebra where all information flow has to be passed explicitly by combinators. The new constructs for specifying information flow and mutually recursive state objects seem to offer a convenient expressive power, but their syntactic manipulation in the reduction semantics may appear complicated. Perhaps a modular structural operational semantics as in [Mos99] would make the formalisation of these constructs more concrete.

The reduction semantics only specifies the possible outcomes of actions, not their divergence behaviour. The specification of divergence is complicated by the concurrent and communicating actions' fairness constraints on progress and message delivery. Possibly, the technique of integer delays used in Mosses' structural operational semantics [Mos92] can be adapted to the communicative part of the reduction semantics.

To assess the laws of the algebra, some form of completeness results should be established. For instance, are the equational laws sufficient to symbolically evaluate some class of first-order actions? Expansion laws have led to completeness results for equational theories of (restricted) process calculi. One can hope for something similar for a suitable first-order fragment of our action algebra. Perhaps the equational completeness result for an imperative language in [MT93] can be adapted to a suitable imperative fragment of our algebra.

The consistency of the algebraic theory remains to be verified. Preferably, its correctness should be established with respect to a notion of operational equivalence—the laws have been formulated with a form of testing equivalence in mind. The results in [Las97] ought to be applicable to the stateless part of the algebra but the remainder is likely to be a formidable task that will need to be approached in stages by developing the necessary theory for suitable fragments.

Acknowledgements I thank Keith Wansbrough for discussions on actions. He and Peter Mosses helpfully commented on drafts of this paper.

References

[DH84] R. DeNicola and M.C.B. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34:83–133, 1984.

[Las94] S. B. Lassen. Design and semantics of action notation. In Peter D. Mosses, editor, *Proc. 1st Intl. Workshop on Action Semantics, Edinburgh, 1994*, number NS-94-1 in BRICS Notes Series, pages 16–33. BRICS, 1994.

[Las97] S. B. Lassen. Action semantics reasoning about functional programs. *Mathematical Structures in Computer Science*, 7(5):557–589, 1997.

[MM93] Martín A. Musicante and Peter D. Mosses. Communicative action notation with shared storage. Tech. Mono. DAIMI PB–452, Dept. of Computer Science, Univ. of Aarhus, 1993.

[Mos83] Peter D. Mosses. Abstract semantic algebras! In Dines Bjørner, editor, *Formal Description of Programming Concepts II.* IFIP, 1983.

[Mos92] Peter D. Mosses. *Action Semantics.* Number 26 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1992.

[Mos96] Peter D. Mosses. Theory and practice of action semantics. In *Proc. 21st Mathematical Foundations of Computer Science, Cracow, Poland*, volume 1113 of *Lecture Notes in Computer Science*, pages 37–61. Springer-Verlag, 1996.

[Mos99] Peter D. Mosses. A Modular SOS for Action Notation, 1999. In this volume.

[MT93] I. A. Mason and C. L. Talcott. Inferring the equivalence of functional programs that mutate data. *Theoretical Computer Science*, 105(2):167–215, 1993.

Tuple Sort Inference in Action Semantics

Kent D. Lee[1,2]

[1] University of Iowa
[2] Luther College
leekentd@luther.edu

Abstract. Sort inference over actions has been studied in the Actress compiler generator. The sort inference algorithm in Actress is based on record sort inference first proposed by Even and Schmidt. However, Actress does not support sort inference over tuples, an important sort in action semantics. In this paper the sort inference algorithm of Even and Schmidt is extended to perform sort inference over tuples. With the development of this algorithm it is now possible to implement a richer and more natural subset of action semantics as compared to the action notation supported by the Actress system. Genesis, a new compiler generator based on Actress, has been developed to demonstrate this new algorithm and is also described in this paper.

1 Introduction

Syntax analysis of programming languages has a well-developed theory that compiler writers take advantage of when constructing parsers and scanners. However, formal semantic descriptions are generally not used in developing compilers. Instead, compiler writers usually rely on English or some other natural language to give the meaning of a programming language. While the English language is well-suited for conversation, its ambiguity causes it to be ineffective in defining programming languages because it may lead to inconsistent interpretations. For example, because the semantics of Pascal was defined informally, the first two compilers for the language treated type equivalence differently: one implemented structural equivalence while the other expected name equivalence, even though Niklaus Wirth himself over-saw the construction of both compilers [12]. Needless to say, mistakes like these are expensive in general and are a good motivation for using unambiguous specifications in compiler design.

Action semantics is one such formal method for language description that has many nice properties. In action semantics, an *action* formally describes the meaning of a source language program. Since action semantics is formally defined, in theory it is possible to automatically generate a compiler for a language given a mapping from its syntax to its action semantics. Action semantics-based compilers and compiler generators are being studied by a small group of people. Research in this area began with the development of action semantics by Peter Mosses [7] and David Watt [12]. In 1991, the first compiler generator based on action semantics was designed by Jens Palsberg in his PhD thesis and was called the Cantor system. Palsberg was assisted by Peter Ørbæk, who implemented the Cantor system in 1991, and a paper [10] about it was published in 1992. Later in 1992, Brown, Moura, and Watt described an action semantics-based compiler generator that they named Actress [3]. In 1993, Bondorf and Palsberg developed another version of the Cantor system based on partial evaluation [1]. Kyung-Goo Doh and David Schmidt developed an action semantics prototyping system that employed many of the same techniques used in automated compiler generation about the same time [4]. Doh and Schmidt's system was partially based on the Actress system. In 1994, Peter Ørbæk enhanced the original Cantor system to develop his own system [8], which he called OASIS.

This paper introduces a new action semantics-based compiler generator called Genesis[6], which is based on the structure of the Actress compiler generator, and discusses its sort inference algorithm.

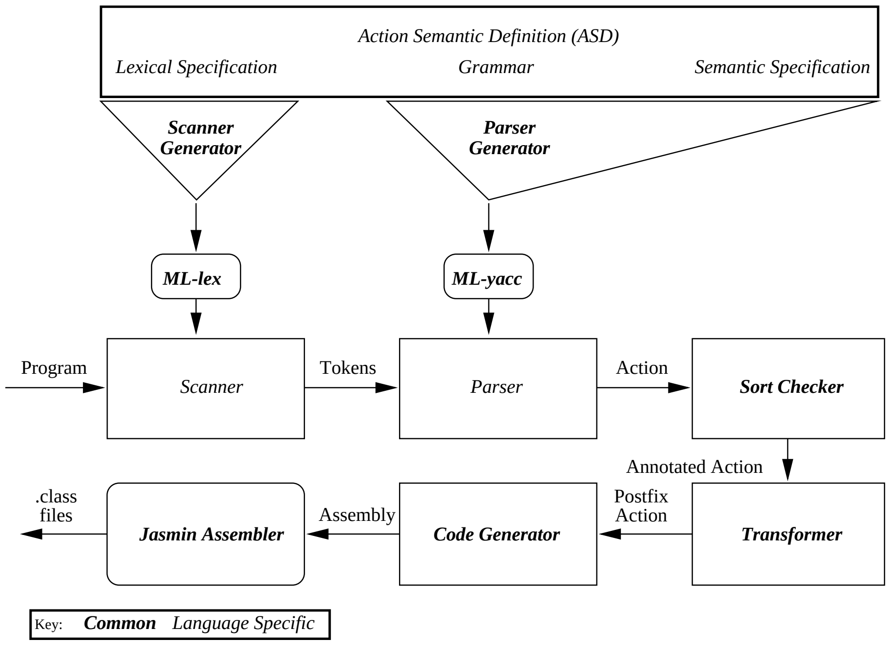


Fig. 1. Structure of Genesis

Figure 1 illustrates the structure of Genesis. An action semantics-based compiler begins by translating a *source program* to its *program action*. The action notation implemented by Genesis is based on the notation introduced in the Actress system. However, Genesis improves on it in several respects. The next section introduces action semantics and describes the action notation supported by Genesis while pointing out a key difference between the action notation supported by Actress and the notation implemented in the Genesis system.

Both Actress and Genesis sort check program actions using a unification-based sort inference algorithm developed by Even and Schmidt [5]. Section 3 describes an algorithm for sort checking tuples in actions including the special tuples called *transients.* While Actress implemented a limited form of sort inference over transients, this paper describes an extension to the sort inference algorithm that enables it to sort check transients and, more generally tuples, in the context of a unification-based sort inference algorithm.

After sort checking an action, an Action Transformer transforms the program action into a postfix form to get it ready for code generation. While Actress generated C target programs, Genesis generates Java Virtual Machine assembly language programs. Transforming the program action to a postfix form gets it ready for code generation targeting low-level languages. Postfix transformations of actions are studied in [6].

To demonstrate the Genesis system a statically typed programming language called Small was developed. Small is a small subset of ML. Small is a block structured language with nested functions of zero or more parameters, functional composition, recursion, and imperative features. For a full description of Small see [6].

2 Action Notation

Like Actress, Genesis supports enough action notation to describe programming languages containing functions of one or more parameters, iteration, selection, and sequential execution. Actress and Genesis support action notation covering the functional, declarative, imperative, and reflective facets. The communicative facet is not considered.

Every action is either a *primitive action* or a *compound action*, composed of other primitive and compound actions that are combined through the use of *combinators.* Combinators dictate how the current information is propagated to their sub-actions. It is useful to study combinators with respect to their facets. Combinators affect how the current information is passed to their sub-actions, and how the information generated by sub-actions is combined. Most combinators are binary, combining two sub-actions into one compound action. For instance, consider the action

```
| give 4
then
| bind "x" to the given value
```

In the functional facet the then combinator propagates the transients given by the first sub-action to the transients used by the second sub-action. So, in this example, (4) is given as the *outgoing* transient of the first sub-action and the second sub-action uses the singleton tuple (4) as its *incoming* transient value.

The behavior of the then combinator is described by figure 2. Combinator diagrams were first introduced by Watt[12] and were revised by Slonneger [11]. They help in understanding the properties of combinators with respect to each

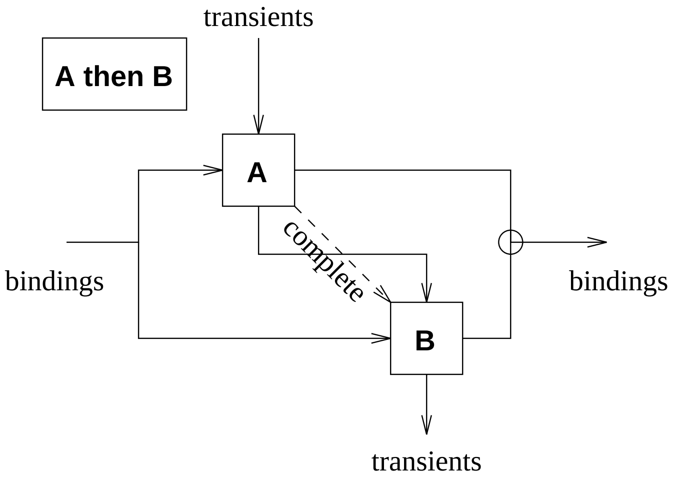


Fig. 2. The then Diagram

facet. The diagrams show the flow of transients and bindings through a compound action with respect to a specific combinator. Transients flow from top to bottom, while bindings flow from left to right. For instance, the then combinator propagates the incoming transients to A. The transients given by A are passed to B, the second sub-action. The transients given by the compound action are the transients given by B. The bindings received by the compound action, A then B, are propagated to each of the sub-actions. Bindings produced by the compound action are the *merged* bindings produced by sub-actions A and B. The dashed line in figure 2 indicates control flow. The then combinator requires that A *completes* first, followed by the performance of B. Since storage is part of the *imperative* facet, changes made to storage by A are seen by B when it is performed.

The behavior of the and then combinator is given in figure 3, which gives the incoming transients to both sub-actions and concatenates the outgoing transient tuples of both sub-actions to form the outgoing transients for the compound action. For instance, the action

```
| give 4
and then
| give 5
```

gives the tuple (4,5). This is a big difference between the action notation supported by Actress and Genesis. In Actress' notation this action would be written

```
| give 4 label 1
and then
| give 5 label 2
```

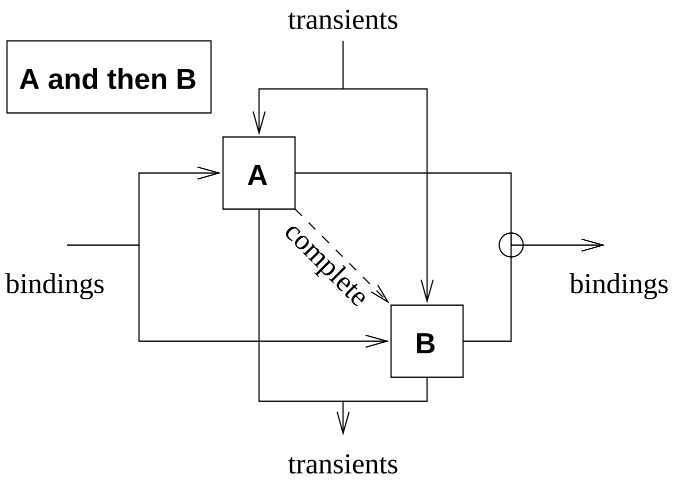


Fig. 3. The and then Diagram

Actress was based on an earlier, working version of action semantics where transients were defined as a map from naturals to sorts. The need to explicitly label positions within the transients means that Actress' notation is not modular, which is one of the goals of action semantics.

Like Actress, Genesis supports the sorts integer, truth-value, cell, and abstraction. Actress supports the list sort. In contrast, Genesis supports the tuple sort. Genesis also supports the use of incomes and outcomes to qualify the sorts of action and yielder.

3 Sort Inference in Action Semantics

The sort inference algorithm used in Genesis is based on the sort inference algorithm used in Actress [2], which is based on the algorithm of Even and Schmidt [5]. The algorithm performs sort unification over records. Records, representing transients and bindings, are unified to infer the sorts of actions, yielders, and data. Genesis extends this algorithm by using records to represent tuples in general and not specifically transients. Unification of two records produces a substitution of *variables* in the two records that must hold for the two records to be identical, or unified. If no substitution is possible, then the two records are not unifiable. The sort inference algorithm presented here is an imperative version of unification. A global substitution is maintained. This is allowable since each variable is unique and is never re-used. The purpose of sort inference is to check that an action is well-formed. An action is well-formed if all references to sorts, either explicitly or implicitly, within the action are consistent.

Actress was the first action semantics-based compiler generator to use Even and Schmidt's sort inference algorithm. The version of the algorithm presented here extends the algorithm in two ways. This implementation includes two constraints as opposed to Actress' algorithm that allowed one type of constraint.

The extension of sort inference to tuples is the other improvement, which has a major impact on the action notation supported by Genesis as compared to Actress. Sort checking of tuples of data has been studied by Ørbæk[9] in his dissertation. However, Ørbæk's approach traces the flow of data over the functional facet through the use of a dependence analysis on the semantic equations of an action semantic definition. His approach does not perform unification of sorts, resulting in difficulties when sort checking actions that contain unfolding/unfold. Ørbæk deals with this by introducing new semantic equations. The new semantic equations become infinite actions that may grow as they are performed much like applying a fix-point. Ørbæk's algorithm for tuple sort inference is sufficiently different from this algorithm that the two are not compared in detail. Ørbæk says that true tuple sort inference cannot be achieved by Even and Schmidt's sort inference algorithm since tuples are represented as records[9]. The algorithm presented here shows how Even and Schmidt's sort inference algorithm can be extended to perform sort inference over tuples. Furthermore, since this algorithm is based on unification of sorts, it handles actions that contain unfolding/unfold. The following sections describe the inference algorithm.

3.1 Sort Schemes

Data Sort	$\sigma \rightarrow$	$\mathsf{nothing} \mid \mathsf{datum} \mid \mathsf{integer} \mid \mathsf{truth\text{-}value} \mid I \mid$
		$[\sigma]\mathsf{cell} \mid \sigma_1 \,\vert\, \sigma_2 \mid \theta \mid \mathsf{abs}(\tau,\beta) \hookrightarrow (\tau',\beta') \mid$
		$(\tau,\beta) \rightsquigarrow \sigma \mid (\tau,\beta) \hookrightarrow (\tau,\beta') \mid \Gamma$
Incoming Transients	$\tau \rightarrow$	Γ
Outgoing Transients	$\tau' \rightarrow$	Γ
Incoming Bindings	$\beta \rightarrow$	Γ
Outgoing Bindings	$\beta' \rightarrow$	Γ
Record	$\Gamma \rightarrow$	$\Gamma_1 \wedge \Gamma_2 \mid \Phi\Psi$
Fields	$\Phi \rightarrow$	$\{id_1 : \phi_1, \cdots, id_n : \phi_n\}$
Field	$\phi \rightarrow$	$\mathsf{absent} \mid \sigma \mid \Delta$
Row	$\Psi \rightarrow$	$\varepsilon \mid \gamma \mid \rho$
Individual	$I \rightarrow$	$\mathsf{false} \mid \mathsf{true} \mid 0 \mid 1 \mid -1 \mid \ldots$

Fig. 4. Sort Schemes in Genesis

Records are structures that are used to represent tuples, transients, and bindings in actions. Records were first introduced by Even and Schmidt[5] to represent the sorts of bindings and transients in actions. Bindings are a map where the domain is a set of identifiers and the range is Sort. In action semantics the transients are a tuple, but are represented as records in the sort inference algorithm. The domain of tuple records are non-negative integers and the range again is

Sort. The first position in a tuple is always mapped to by 1, the next by 2, and so on. Sort checking actions involves unifying records, which implies that there are some sorts in the two records that are unknown. Unknown information is represented by variables. Sorts containing variables are called sort schemes and their syntax is given in figure 4. I represents all individuals. Γ represents records. A tuple also may be constructed by concatenating two tuples as in $\Gamma_1 \wedge \Gamma_2$ in figure 4. Tuple concatenation is a new addition to the sort inference algorithm and is described in more detail in the next sections.

At this point it is important to notice that records are also sorts. Previous versions of this algorithm distinguished between records and sorts. In this version of the sort inference algorithm steps are taken to consider tuples, and therefore records, themselves as sorts. Allowing records to be considered sorts has implications that must be considered and dealt with.

Cell sorts are qualified by a sort that may be stored in them because target languages typically have different storage requirements for storing an integer as opposed to a truth-value for instance. There are infinitely many cell sorts, but only a few are of interest. In particular, integer and truth-value cells are of interest in the Small language. However, other sorts of cells can be constructed to suit the needs of other languages. Nothing in the sort inference algorithm precludes other kinds of cell sorts.

The productions for incoming and outgoing transients and bindings are not needed by the grammar but are included here because it is convenient to use τ,β,τ',β' in the sort inference rules presented in appendix A. The sort $(\tau, \beta) \hookrightarrow (\tau', \beta')$ represents the sort of actions. An action is a function of the current information. In the case of sort inference the current information is the transients and bindings that are given to an action. Actions produce new bindings and give new transients when performed. τ and β are the incoming transients and bindings and τ' and β' are the outgoing transients and bindings, respectively. The sort of yielders is represented by $(\tau, \beta) \rightsquigarrow \sigma$. Yielders may also be a function of the current information and they yield data, which is given by σ. The sorts of abstractions are represented like actions since they affect the current information in the same way when they are enacted. The $\mathsf{abs}(\tau, \beta) \hookrightarrow (\tau', \beta')$ notation represents the sort of an abstraction in the sort inference algorithm.

4 Unification of Sort Schemes

In Genesis, two sort schemes are unified according to the algorithm presented in figure 5, which is written in the ML programming style. ML implements pattern-matching, which is used extensively in this algorithm description to conserve space. Pattern-matching proceeds from top to bottom, matching the first correct pattern for the two sorts being unified. Many details of the algorithm are omitted here. More details can be found in [6]. Unification of two sort schemes finds a substitution for the variables in the two sorts. There are three kinds of variables in sort schemes.

– Sort Variables

```
       $unify : \sigma \rightarrow \sigma \rightarrow \sigma$ (perhaps altering the global subsitution)
(1)      $unify\ \theta_i\ \theta_j = ([\theta_i \mapsto \theta, \theta_j \mapsto \theta];\ \theta)$
(2)  |   $unify\ \theta_i\ \sigma_j = ([\theta_i \mapsto \sigma_j];\ \theta_i)$
(3)  |   $unify\ \sigma_i\ \theta_j = unify\ \theta_j\ \sigma_i$
(4)  |   $unify\ (\{\} \wedge \sigma_j)\ \sigma_k = unify\ \sigma_j\ \sigma_k$
(5)  |   $unify\ (\sigma_i \wedge \{\})\ \sigma_k = unify\ \sigma_i\ \sigma_k$
(6)  |   $unify\ (\Phi_i \wedge \Phi_j)\ \sigma_k = unify\ (\Phi_i \cdot \Phi_j)\ \sigma_k$
(7)  |   $unify\ (\Phi_i\Psi_i \wedge \Phi_j\Psi_j)\ \Phi_k =$
             if (length($\Phi_i \cdot \Phi_j$) = length($\Phi_k$) then ($[\Psi_i \mapsto \{\}, \Psi_j \mapsto \{\}];\ unify\ (\Phi_i \cdot \Phi_j)\ \Phi_k$)
             else raise $concatFailure$
(8)  |   $unify\ (\Gamma_i \wedge \Gamma_j)\ \Gamma_k =$ raise $concatFailure$
(9)  |   $unify\ \sigma_i\ (\sigma_j \wedge \sigma_k) = unify\ (\sigma_j \wedge \sigma_k)\ \sigma_i$
(10) |   $unify\ (\sigma_{i_1} \mid \cdots \mid \sigma_{i_n})\ (\sigma_{j_1} \mid \cdots \mid \sigma_{j_m}) =$
             let val $m = (\sigma_{i_1} \mid \cdots \mid \sigma_{i_n})\ \&\ (\sigma_{j_1} \mid \cdots \mid \sigma_{j_m})$
                 val $\sigma'_i = prune\ (unify\ \sigma_{i_1} m \mid \cdots \mid unify\ \sigma_{i_n} m)$
             val $\sigma'_j = prune\ (unify\ \sigma_{j_1} m \mid \cdots \mid unify\ \sigma_{j_n} m)$ in
                 case $(\sigma'_i, \sigma'_j)$ of
                     $(\sigma'_{i_1} \mid \cdots \mid \sigma'_{i_g}, \sigma'_{j_1} \mid \cdots \mid \sigma'_{j_h}) \Rightarrow |_{k=1,l=1}^{g,h}\ unify\ \sigma_{i_k}\sigma_{j_l}$ where $\sigma_{i_k} \& \sigma_{j_l} \neq nothing$
                 |   $(\_,\_) \Rightarrow unify\ \sigma'_i\sigma'_j$ end
(11) |   $unify\ (\sigma_{i_1} \mid \cdots \mid \sigma_{i_n})\ \sigma_j =$
             let val $m = (\sigma_{i_1} \mid \cdots \mid \sigma_{i_n})\ \&\ \sigma_j$ in
                 $unify\ \sigma_j m;\ \forall k\ unify\ \sigma_{i_k} m;\ \forall k\ unify\ \sigma_{i_k}\sigma_j$ where $\sigma_{i_k} \& \sigma_j \neq$ nothing;
                 $prune\ (\sigma_{i_1} \mid \cdots \mid \sigma_{i_n})$ end
(12) |   $unify\ \sigma_i\ (\sigma_{j_1} \mid \cdots \mid \sigma_{j_m}) = unify\ (\sigma_{j_1} \mid \cdots \mid \sigma_{j_m})\ \sigma_i$
(13) |   $unify\ [\sigma_i]$cell $[\sigma_j]$cell $= [unify\ \sigma_i\sigma_j]$cell
(14) |   $unify\ \Gamma_i\ \Gamma_j = combine\ unifyField\ unifyRow\ \Gamma_i\Gamma_j$
(15) |   $unify$ abs$(\tau_i, \beta_i) \hookrightarrow (\tau'_i, \beta'_i)$ abs$(\tau_j, \beta_j) \hookrightarrow (\tau'_j, \beta'_j) =$
             abs$(unify\tau_i\tau_j, unify\beta_i\beta_j) \hookrightarrow (unify\tau'_i\tau'_j, unify\beta'_i\beta'_j)$
(16) |   $unify\ I$ integer $= I$, if $I <$integer
(17) |   $unify\ I$ truth-value $= I$, if $I <$truth-value
(18) |   $unify\ \sigma\ I = unify\ I\ \sigma$
(19) |   $unify\ I_1\ I_2 = I_1$ if $I_1 = I_2$ else nothing
(20) |   $unify\ I$ datum $= I$
(21) |   $unify$ datum $I = I$
(22) |   $unify$ integer integer = integer
(23) |   $unify$ truth-value truth-value = truth-value
(24) |   $unify\ [\sigma_i]$cell $\sigma_j = (unify\ \sigma_i$ nothing; nothing)
(25) |   $unify\ \sigma_i\ [\sigma_j]$cell $= (unify\ \sigma_j$ nothing; nothing)
(26) |   $unify\ \sigma_i\ \sigma_j =$ nothing
```

Fig. 5. The *unify* Operation

Sort variables, θ, represents an unknown sort that may be bound to any σ.

– Field Variables

  Field variables, $\varDelta$, represent fields in a record that are either present (i.e. may be bound to some σ) or absent, but it is unknown which is true.

– Row Variables

  γ variables represents unknown fields in a record that are ignored and ρ variables represent information that is propagated by an action or yielder.

The symbol ε represents the absence of a row variable, meaning the record does not contain any additional fields that are not represented in the $\varPhi$ part of the record. Records that do not contain a row variable usually omit the ε. In figure 5, sort variables are assumed to be unbound. If a sort variable, θ_i, were bound to a sort, σ_i, then the substitution could be applied to unify σ_i instead.

The result of a successful unification of sorts is a (possible) change in the global substitution and a sort. Unifications in figure 5 that cause a modification of the global substitution show the effects of the alteration inside of brackets. For instance, in the first unification unbound θ_i and θ_j are bound to a new sort variable θ as a result of unifying the two unknown sorts and the result of the unification is θ.

In general, if a valid substitution is found, then unification is successful. If no substitution exists, then unification algorithms normally report failure. This is not the case in action semantics. Action semantics is based on the principle of sorts, not types. There is a complete partial ordering of sorts in action semantics. This means that two sort schemes in action semantics are always unifiable. It is trivial to report that unification always yields the sort nothing, which is the sort that appears at the bottom of the complete partial ordering of sorts. A unification algorithm that always returns nothing, while much simpler than the algorithm presented here, is uninteresting! The goal of unification in action semantics is to find a substitution that results in the greatest lower bound of the two sort schemes being unified. As such, the $unify$ operation given in figure 5 returns nothing as a last resort instead of reporting failure.

Unification of tuples is complex enough to warrant some further explanation. Equations 4-7 represent unification between two sorts, one of which was constructed by concatenating two tuples (i.e. $\sigma_i \wedge \sigma_j$). Tuple concatenation can only be carried out in certain restricted cases. Equations 4-6 are straightforward applications of equations involving tuple concatenation. If the size of the resulting tuple is known, or if one tuple is empty, tuple concatenation is possible. Equation 7 states that a concatenated tuple of unknown size, $\varPhi_i\varPsi_i \wedge \varPhi_j\varPsi_j$, may be unified with another sort, $\varPhi_k$, of known size if the number of known fields in the concatenated tuple, length$(\varPhi_i \cdot \varPhi_j)$, is equal to the number of fields that are expected in $\varPhi_k$. Tuple concatenation should not be carried out if the size of the resulting tuple is unknown. For instance, the only possible unification of $\varPhi_i\gamma_i \wedge \varPhi_j\gamma_j$ and $\{\}\gamma_k$ would result in nothing since the size of the resulting tuple is unknown due to the unbound row variables.

5 Unifying Records

Combinators like `then` and `and then` dictate how transients and bindings flow through an action. For instance, from figure 3 it is easy to see that the incoming transients are given to both sub-actions and the outgoing transients of the entire action are comprised of the concatenation of the outgoing transients of the two sub-actions. The `and then` combinator dictates tuple concatenation be used to arrive at the appropriate outgoing transients for the entire action. In other words, arising from the intended behavior of action semantic combinators are a set of *derived* sort operations over the sorts of incoming and outgoing transients and bindings. In this version of the sort inference algorithm, six derived sort operations are described. Brown named five of these operations in his version of the sort inference algorithm[2]. This work has led to the discovery of one more for tuple concatenation.

The derived sort operators can be visualized by looking at Slonneger's combinator diagrams[11]. For instance, consider the `and then` combinator in figure 3. In this diagram there are three different derived sort operators presented. They are $unify$, *concat*, and *merge*. The same incoming transients are given to each of the sub-actions. Unification is the appropriate sort operator to use in this case. Brown distinguishes between records and sorts in his sort inference algorithm. As a result, he also distinguishes between unification of sorts and unification of records. He calls the unification of records the *distribute* operator in his algorithm. The *distribute* operator in the Actress system and the $unify$ operation over records in Genesis are handled identically. The incoming bindings are also given to each of the sub-actions and $unify$ is again the sort operator for the incoming bindings to the `and then` combinator.

The outgoing bindings of the two sub-actions are merged to form the outgoing bindings of the entire action. The *merge* operator insists the bindings be disjoint. The two sub-actions may not bind the same identifier. The outgoing transients of the two sub-actions are concatenated together. Tuple concatenation is represented by the *concat* sort operator.

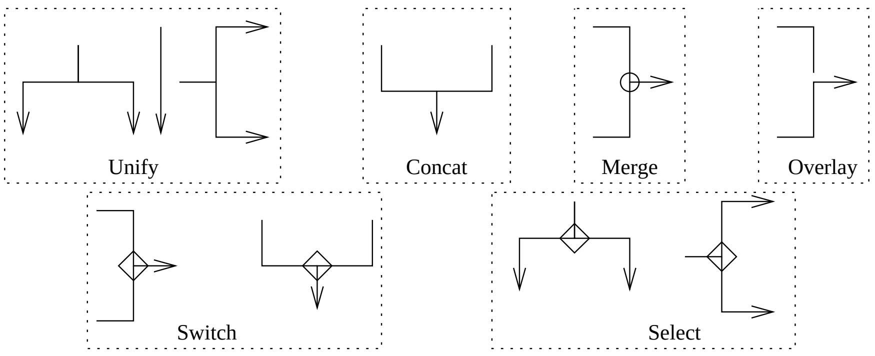


Fig. 6. Combinator Operators

There are three other derived sort operators that arise from action semantics and appear in combinator diagrams presented in [6]. Each sort operation combines two record sorts in different ways. These operators are used in the sort inference rules presented in appendix A and each of the operators are graphically depicted in figure 6 and described informally here.

- *unify*
  The *unify* operation is simply unification over sorts. There is no distinction made in this algorithm between unification of sorts and unification of records, since records are themselves sorts. This operator is applied to both incoming and outgoing transients and bindings.
- *concat*
  This operator denotes tuple concatenation and is used to concatenate outgoing transients of sub-actions for the appropriate combinators. In addition, this operator is used to implement sort inference over tuples.
- *merge*
  The *merge* sort operator merges two disjoint maps. *merge* is used to represent the merging of outgoing bindings from actions.
- *overlay*
  The *overlay* operation is a union of two maps where the bindings of the second subaction take precedence over bindings of the same identifiers produced by the first subaction. Scoped variables in a program are obtained by overlaying one binding map on another.
- *switch*
  The *switch* operator is used to choose between one of two possibilities for incoming transients and bindings. It is used by the or and else combinators.
- *select*
  The *select* operator models the choice of sorts that can be given or produced from the outgoing transients or bindings of the two sub-actions of an action involving the or or else combinator.

Each sort operator combines the fields and row variables of two records in a unique way. Details of how all but the *concat* operator work are ommitted here, but can be found in [6].

6 Constraints

The sort inference rules presented in appendix A contain a few constraints, which are restrictions on sorts that must be enforced at the end of the sort inference algorithm. Brown defines one type of constraint in the Actress sort inference algorithm, namely to check $\theta\&\sigma\neq\mathsf{nothing}$ for some bound sort variable θ and sort σ. In this version of the algorithm, constraint checking is expanded to two different types of constraints. The first of these is called a sub-sort constraint, stated $\sigma_1 \leq \sigma_2$, which means at the end of the sort inference σ_1 must be less than or equal to σ_2 in terms of the complete partial ordering of sorts. The $\sigma_1 \leq \sigma_2$ constraint has slightly different semantics than a $\theta\&\sigma\neq\mathsf{nothing}$ constraint. The sub-sort constraint is advantageous in sort checking several rules[6]. For instance, the sub-sort constraint allows sort checking of polymorphic abstractions and the is yielder. While Actress' constraint is most general, it sometimes leads to run-time sort checks. One goal of Genesis is to remove all run-time sort checks.

$$\begin{array}{ll} & \mathit{concat}\ (\Phi_i \rho_i)\ \Gamma_j = (\mathit{unity}((\Phi_i \rho_i) \wedge \Gamma_j, \{\}\rho);\ \{\}\rho) \\ | & \mathit{concat}\ \Gamma_i\ (\Phi_j \rho_j) = (\mathit{unity}(\Gamma_i \wedge (\Phi_j \rho_j), \{\}\rho);\ \{\}\rho) \\ | & \mathit{concat}\ \Gamma_i\ \Gamma_j = (\mathit{unity}(\Gamma_i \wedge \Gamma_j, \{\}\gamma);\ \{\}\gamma) \end{array}$$

Fig. 7. The *concat* Operator

Genesis' version of the sort inference algorithm introduces one more type of constraint, called a unity constraint. Unity constraints are actually unifications that are to be performed in the second stage of the sort inference algorithm. They currently have one purpose, which is to assist in implementing the *concat* operator, as given in figure 7. The *unity* function imperatively adds a unity constraint to the unity constraint list but does not attempt any unification, which is an important distinction from the $unify$ sort operator, since tuple concatenations are not always possible during the first stage of the sort inference algorithm. Instead, the *unity* constraint declares an intention to $unify$ in the second stage of the algorithm. How unity constraints are satisfied is described in the next section.

7 The Sort Inference Algorithm

The sort inference algorithm consists of three stages: annotating the action, satisfying constraints, and reducing the sorts. The following sections describe what happens during each stage. The algorithm is given a program action with no sort annotations and either ends in sort inference failure or succeeds in annotating the program action and its sub-actions with sort information.

7.1 Annotating the Action

The first stage compositionally annotates the program action and its sub-actions with sort information as dictated by the sort inference rules in appendix A. Sort operators are applied to records, constraints are added to the subsort and unity constraint lists, and sort, row, and field variables are bound to sorts schemes, field schemes, and record schemes in the global substitution, respectively.

7.2 Satisfying Constraints

Constraint checking is performed during the second stage. This stage starts by looping over the list of subsort and unity constraints, attemptying to satisfy them. Every pass of the loop performs each constraint operation, refining sorts that are involved in the constraints. If failure occurs during constraint satisfaction, the sort inference algorithm fails. It is possible that unification of two sorts in a unity constraint could result in *concatFailure*, indicating that there is insufficient information to concatenate two tuples that are referred to in a unity constraint. *concatFailure* does not result in sort inference failure or return nothing since tuple concatenation may succeed on a subsequent iteration. The loop continues until a complete pass produces no further refinements in the sorts being constrained. The loop is guaranteed to terminate since a canonical form exists for sorts. At some point, each constraint will either be satisfied, will result in failure, or will result in *concatFailure*.

Following the constraint loop, any remaining unbound sort, field, or row variables are instantiated to datum, absent, and {}, respectively. Finally, each constraint is performed one more time to refine any sorts that might be further refined due to instantiating the unbound variables.

7.3 Reducing the Sorts

At this point, the sort inference algorithm has succeeded in assigning a sort to the program action and its sub-actions. However, there are typically many bound sort, field, and row variables that form long substitution chains. The last stage of the algorithm applies the variable substitution to remove all non-essential variables from the sorts. All field and row variables are removed. All absent fields in records are also eliminated. All sort variables except those bound directly to a sort, and not another sort variable, are removed from the sorts of the action. The remaining sort variables represent information that is needed by the action transformer.

8 An Example of Sort Inference

This section presents an example of sort inference over tuples. Consider the Small program

```
let fun f(x,y) = x+y
in
  output(f(4,5))
end
```

This program can be translated to the action according to the action semantic description found in [6]. Tuples occur in two places in the program's action. The first is where 4 and 5 are passed to the function.

```
| give 4 and then give 5
then
| enact application of the abstraction
| bound to "f" to the given data
```

The second occurrence is where the two parameters are given to the function.

```
bind "f" to closure of the abstraction of
| | furthermore
| | | | | bind "x" to the given (integer|truth-value)#1
| | | | and then
| | | | | give the rest of the given data
| | | then
| | | | bind "y" to the given (integer|truth-value)
| thence ...
```

To annotate the first action the two sub-actions 'give 4' and 'give 5' are annotated using the INDIVIDUAL and GIVE-INDIVIDUAL rules as in

give 4
: $(\{\}\gamma_{154},\{\}\gamma_{155}) \hookrightarrow (\{1{:}\theta_{45}\},\{\})$

give 5
: $(\{\}\gamma_{156},\{\}\gamma_{157}) \hookrightarrow (\{1{:}\theta_{46}\},\{\})$

$[\theta_{45} \mapsto 4, \theta_{46} \mapsto 5]$

with the given substitution. Next, the sorts of the two sub-actions are combined as dictated by the AND-THEN rule to annotate the whole action

```
| give 4
and then
| give 5
```
: $(\{\}\gamma_{154},\{\}\gamma_{155}) \hookrightarrow (\{\}\gamma_{158},\{\})$

$[\gamma_{156} \mapsto \{\}\gamma_{154}]$
constraint: unity$(\{\}\gamma_{158},\{1{:}\theta_{45}\}\wedge\{1{:}\theta_{46}\})$

At this point a unity constraint is introduced by the *concat* operation in the AND-THEN sort inference rule. There is enough information to carry out tuple concatenation, but the algorithm does not attempt to do the concatenation now. Instead it introduces the unity constraint and represents the concatenated tuple as $\{\}\gamma_{158}$.

To annotate the other action the GIVEN#, BIND-TO, REST, and GIVE-DATA rules are applied to annotate these two sub-actions

bind "x" to the given (integer|truth-value)#1
: $(\{1{:}\theta_{12}\}\gamma_{59},\{\}\gamma_{60}) \hookrightarrow (\{\},\{\mathsf{x}{:}\theta_{12}\})$

give the rest of the given data
: $(\{\}\rho_{71},\{\}\gamma_{69}) \hookrightarrow (\{\}\rho_{70},\{\})$

$[\theta_{12} \mapsto$ (integer|truth-value)$, \theta_{13} \mapsto$ datum$]$
constraint: unity$(\{1{:}\theta_{13}\}\wedge\{\}\rho_{70},\{\}\rho_{71})$

Another unity constraint is introduced by the *concat* invocatoin in the REST rule. This is an instance where the power of a unification based algorithm is needed. In this case, the *concat* operator is used to reverse the effects of tuple concatenation by selecting all but the first element of the incoming transients. The compound action is annotated using the AND-THEN rule

```
| bind "x" to the given (integer|truth-value)#1
and then
| give the rest of the given data
```
: $(\{1{:}\theta_{12}\}\rho_{72},\{\}\gamma_{60}) \hookrightarrow (\{\}\rho_{73},\{x{:}\theta_{12}\})$

$[\gamma_{59} \mapsto \{\}\rho_{72}, \rho_{71} \mapsto \{1{:}\theta_{12}\}\rho_{72}]$
constraint: unity$(\{\}\wedge\{\}\rho_{70},\{\}\rho_{73})$

Next, the BIND-TO and GIVEN rules are now used to annotate the action that binds "y".

bind "y" to the given (integer|truth-value)
: $(\{1{:}\theta_{16}\},\{\}\gamma_{75}) \hookrightarrow (\{\},\{y{:}\theta_{16}\})$

$[\theta_{16} \mapsto$(integer|truth-value)$]$

And finally, the entire action can be annotated using the THEN rule

```
| | bind "x" to the given (integer|truth-value)#1
| and then
| | give the rest of the given data
then
| bind "y" to the given (integer|truth-value)
```
: $(\{1{:}\theta_{12}\}\rho_{72},\{\}\gamma_{60}) \hookrightarrow (\{\},\{x{:}\theta_{12},y{:}\theta_{16}\})$

$[\rho_{73} \mapsto \{1{:}\theta_{16}\}]$

The rest of the action is annotated as well, but is ommitted.

Once annotation is complete, constraint satisfaction begins. After applying the substitution, the three unity constraints are

1. unity$(\{\}\gamma_{158},\{1{:}\theta_{45}\}\wedge\{1{:}\theta_{46}\})$
2. unity$(\{1{:}\theta_{13}\}\wedge\{\}\rho_{70},\{1{:}\theta_{12}\}\rho_{72})$
3. unity$(\{\}\wedge\{\}\rho_{70},\{1{:}\theta_{16}\})$

$[\theta_{12} \mapsto$(integer|truth-value)$,\theta_{13} \mapsto$datum,
$\theta_{16} \mapsto$(integer|truth-value)$,\theta_{45} \mapsto 4,\theta_{46} \mapsto 5]$

During the first pass of constraint satisfaction, the first and third tuple concatenations can be performed immediately. Attempting to unify the two sorts in the second constraint results in *concatFailure* according to the algorithm for $unify$ in figure 5. The first pass of constraint satisfaction alters the substitution to

$[\gamma_{158} \mapsto \{1{:}\theta_{45},2{:}\theta_{46}\},\rho_{70} \mapsto \{1{:}\theta_{16}\}]$

The remaining unity constraint can now be updated by applying the new substitution. The resulting constraint is unity($\{1{:}\theta_{13}\}\wedge\{1{:}\theta_{16}\}$,$\{1{:}\theta_{12}\}\rho_{72}$). At this point the final tuple concatenation can be performed resulting in the final change in the substitution

$[\theta_{12} \mapsto \theta_{50}, \theta_{13} \mapsto \theta_{50}, \theta_{50} \mapsto$(integer|truth-value)$, \rho_{72} \mapsto \{1{:}\theta_{16}\}]$

Applying the final substitution the sorts of the two actions can be determined. The final annotated actions are

```
| give 4
and then
| give 5
: ({}γ154,{}γ155) ↪ ({1:θ45,2:θ46},{})
```

and

```
| | bind "x" to the given (integer|truth-value)#1
| and then
| | give the rest of the given data
then
| bind "y" to the given (integer|truth-value)
: ({1:θ50,2:θ16},{}γ60) ↪ ({},{x:θ50,y:θ16})
```

Since there was a change in the substitution on the second pass of constraint satisfaction, a third pass over the constraints would be attempted. On the third pass no further constraint satisfaction (i.e. no changes in the substitution) would occur and the constraint satisfaction loop terminates. The sorts are reduced by applying the final substitution to the annotated action. In Genesis, the annotated action becomes the input to the action transformer, which transforms the action to postfix form. The postfix action serves as input to the code generator, which generates a Java Virtual Machine program.

9 Conclusion

Previously, it was thought that sort inference over tuples was not possible in the context of Even and Schmidt's algorithm[5]. This paper has demonstrated an extension to the algorithm that implements sort inference over tuples in action semantics. In addition, a set of sort inference rules allowing the expression of tuples in actions is given in appendix A.

The existence of an algorithm to perform tuple sort inference in action semantics means a truer subset of action notation can be implemented. Where the action notation implemented by the Actress system was not modular[3], the notation presented here does not have to label positions within transient values explicitly. The result is modular action semantic definitions that can be re-used easily.

Tuples are also valuable when passing parameters to abstractions. To give data to abstraction enactions, the Actress system requires the data be packaged in the form of a list, resulting in the possibility of lost sort information. Using the algorithm presented here, data can be given to abstractions in the form of a tuple, which does not result in a loss of sort information.

Finally, this paper provides an overview of a new compiler generator based on action semantics called Genesis. Genesis builds on the work of the Actress project and serves as a demonstration of the sort inference algorithm presented in this paper.

A Action Sort Inference Rules

A.1 Rule Format

The set of representable sort shemes are given by the grammar presented in figure 4. A complete set of the sort inference rules of Genesis is given in [6].

The sort inference rules presented below have the format:

(Rule Name)

$$\frac{\mathit{Antecedent}_1; \mathit{Antecedent}_2;\ ...; \mathit{Antecedent}_n}{\mathit{Conclusion}}$$

Each rule has a conclusion that is satisfied assuming each antecedent is satisfied. Antecedents may be an inference of the form $\varepsilon \vdash \alpha : \sigma$, a record sort expression of the form $\Gamma \neq \mathsf{nothing}$ involving one or more of the record unifiers presented in section 5, a constraint of the form $\sigma_1 \leq \sigma_2$, or a binding of the form $\theta \& \sigma \neq \mathsf{nothing}$. The list of antecedents may be empty. Quantification of free variables occurring in the antecedents are omitted since all variables are universally quantified.

A.2 Rules

(And-Then)

$$\frac{\varepsilon \vdash a_1 : (\tau_1, \beta_1) \hookrightarrow (\tau_1', \beta_1'); \varepsilon \vdash a_2 : (\tau_2, \beta_2) \hookrightarrow (\tau_2', \beta_2')}{\varepsilon \vdash a_1 \ \mathsf{and\ then}\ a_2 : (\mathit{unify}\ \tau_1\tau_2, \mathit{unify}\ \beta_1\beta_2) \hookrightarrow (\mathit{concat}\ \tau_1'\tau_2', \mathit{merge}\ \beta_1'\beta_2')}$$

(Then)

$$\frac{\varepsilon \vdash a_1 : (\tau_1, \beta_1) \hookrightarrow (\tau_1', \beta_1'); \varepsilon \vdash a_2 : (\tau_2, \beta_2) \hookrightarrow (\tau_2', \beta_2'); \mathit{unify}\ \tau_1'\tau_2 \neq \mathsf{nothing}}{\varepsilon \vdash a_1 \ \mathsf{then}\ a_2 : (\tau_1, \mathit{unify}\ \beta_1\beta_2) \hookrightarrow (\tau_2', \mathit{merge}\ \beta_1'\beta_2')}$$

(Furthermore)

$$\frac{\varepsilon \vdash a : (\tau, \beta) \hookrightarrow (\tau', \beta')}{\varepsilon \vdash \mathsf{furthermore}\ a : (\tau, \mathit{unify}\ \beta\{\}\rho) \hookrightarrow (\tau', \mathit{overlay}\ \beta'\{\}\rho)}$$

(Bind-To)

$$\frac{\varepsilon \vdash y : (\tau, \beta) \rightsquigarrow \sigma}{\varepsilon \vdash \mathsf{bind}\ \mathit{id}\ \mathsf{to}\ y : (\tau, \beta) \hookrightarrow (\{\mathit{id} : \sigma\}, \{\})}$$

(Enact)

$$\frac{\varepsilon \vdash y : (\tau, \beta) \rightsquigarrow (\mathsf{abs}(\{\}, \{\}) \hookrightarrow (\tau', \beta'))}{\varepsilon \vdash \mathsf{enact}\ y : (\tau, \beta) \hookrightarrow (\tau', \beta')}$$

(Give-Individual)

$$\frac{\varepsilon \vdash y : (\tau,\beta) \rightsquigarrow \sigma; \sigma \& I \neq \textsf{nothing}}{\varepsilon \vdash \textsf{give}\ y : (\tau,\beta) \hookrightarrow (\{1:\sigma\},\{\})}$$

(Give-Data)

$$\frac{\varepsilon \vdash y : (\tau,\beta) \rightsquigarrow \sigma; \sigma \& \Gamma \neq \textsf{nothing}}{\varepsilon \vdash \textsf{give}\ y : (\tau,\beta) \hookrightarrow (\sigma,\{\})}$$

(Individual)

$$\frac{\varepsilon \vdash I : \sigma; \theta \& \sigma \neq \textsf{nothing}}{\varepsilon \vdash I : (\{\}\gamma_1,\{\}\gamma_2) \rightsquigarrow \theta}$$

(Abstraction)

$$\frac{\varepsilon \vdash a : (\tau,\beta) \hookrightarrow (\tau',\beta'); \theta \& \textsf{abs}(\tau,\beta) \hookrightarrow (\tau',\beta') \neq \textsf{nothing}}{\varepsilon \vdash \textsf{abstraction of}\ a : (\{\}\gamma_1,\{\}\gamma_2) \rightsquigarrow \theta}$$

(Closure)

$$\frac{\varepsilon \vdash y : (\tau,\beta) \rightsquigarrow \textsf{abs}(\tau_a,\beta_a) \hookrightarrow (\tau'_a,\beta'_a); \theta \& \textsf{abs}(\tau_a,\{\}) \hookrightarrow (\tau'_a,\beta'_a) \neq \textsf{nothing}}{\varepsilon \vdash \textsf{closure}\ y : (\tau, unify\ \beta\beta_a) \rightsquigarrow \theta}$$

(Application)

$$\frac{\begin{array}{c}\varepsilon \vdash y_1 : (\tau_1,\beta_1) \rightsquigarrow \sigma_1; \varepsilon \vdash y_2 : (\tau_2,\beta_2) \rightsquigarrow \sigma_2; \textsf{abs}(\tau_a,\{\}) \hookrightarrow (\tau'_a,\{\}) \leq \sigma_1;\\ unify\ \sigma_2\tau_a \neq \textsf{nothing}; \theta \& \textsf{abs}(\{\},\{\}) \hookrightarrow (\tau'_a,\{\}) \neq \textsf{nothing}\end{array}}{\varepsilon \vdash \textsf{application of}\ y_1\ \textsf{to}\ y_2 : (unify\ \tau_1\tau_2, unify\ \beta_1\beta_2) \rightsquigarrow \theta}$$

(Rest)

$$\frac{\varepsilon \vdash y : (\tau,\beta) \rightsquigarrow \sigma; \theta \& datum \neq \textsf{nothing}; unify\ \sigma(concat\ \{1:\theta\}\{\}\rho) \neq \textsf{nothing}}{\varepsilon \vdash \textsf{rest}\ y : (\tau,\beta) \rightsquigarrow \{\}\rho}$$

(Given-Data)

$$\frac{}{\varepsilon \vdash \textsf{given data} : (\tau,\beta) \rightsquigarrow \tau}$$

(Given)

$$\frac{\varepsilon \vdash s : \sigma; \theta \& \sigma \neq \textsf{nothing}}{\varepsilon \vdash \textsf{given}\ s : (\{1:\theta\},\{\}\gamma_1) \rightsquigarrow \theta}$$

(Given#)

$$\frac{\varepsilon \vdash s : \sigma; \theta \& \sigma \neq \textsf{nothing}}{\varepsilon \vdash \textsf{given}\ s\#n : (\{n:\theta\}\gamma_1,\{\}\gamma_2) \rightsquigarrow \theta}$$

(Bound-To)

$$\frac{\varepsilon \vdash s : \sigma; \theta \& \sigma \neq \textsf{nothing}}{\varepsilon \vdash s\ \textsf{bound to}\ id : (\{\}\gamma_1,\{id:\theta\}\gamma_2) \rightsquigarrow \theta}$$

(Sum)

$$\frac{\begin{array}{c}\varepsilon \vdash y : (\tau,\beta) \rightsquigarrow \sigma; \theta_1,\theta_2,\theta_3 \& \textsf{integer} \neq \textsf{nothing};\\ unify\ \sigma\{1:\theta_1, 2:\theta_2\} \neq \textsf{nothing}; \theta_3 \leq \lceil(\theta_1 \mid \theta_2)\rceil\end{array}}{\varepsilon \vdash \textsf{sum}\ y : (\tau,\beta) \rightsquigarrow \theta_3}$$

(Tuple)

$$\frac{\varepsilon \vdash y_i : (\tau_i,\beta_i) \rightsquigarrow \sigma_i; unify\ \tau_i\{\}\gamma_1 \neq \textsf{nothing}; unify\ \beta_i\{\}\gamma_2 \neq \textsf{nothing}}{\varepsilon \vdash (y_1,\cdots,y_n) : (\{\}\gamma_1,\{\}\gamma_2) \rightsquigarrow concat\ \Gamma_1\cdots\Gamma_n; \Gamma_i = \sigma_i\ or\ \Gamma_i = \{1:\sigma_i\}}$$

References

1. A. Bondorf and J. Palsberg. Compiling actions by partial evaluation. In *Proceedings of Conference on Functional Programming Languages and Computer Architecture (FCPA '93)*, Copenhagen, DK, 1993.
2. D.F. Brown. *Sort Inference in Action Semantics.* PhD thesis, Department of Computer Science, University of Glasgow, 1996.
3. D.F. Brown, H. Moura, and D.A. Watt. Actress: an action semantics directed compiler generator. In *Proceedings of the Workshop on Compiler Construction*, Paderborn, Germany, 1992.
4. K.G. Doh and D.A. Schmidt. Action semantics-directed prototyping. *Computer Languages*, Vol.19, No. 4:213–233, 1993.
5. S. Even and D.A. Schmidt. Type inference for action semantics. In *ESOP '90, 3rd European Symposium on Programming, volume 432 of Lecture Notes in Computer Science*, pages 118–133, Berlin, Germany, 1990. Springer-Verlag.
6. K.D. Lee. *Action Semantics-based Compiler Generation (forthcoming).* PhD thesis, Department of Computer Science, University of Iowa, 1999.
7. P.D. Mosses. *Action Semantics: Cambridge Tracts in Theoretical Computer Science 26.* Cambridge University Press, 1992.
8. P. Ørbæk. Oasis: An optimizing action-based compiler generator. In *Proceedings of the International Conference on Compiler Construction, Volume 786*, Edinburgh, Scotland, 1994. LNCS.
9. P. Ørbæk. *Trust and Dependence Analysis.* PhD thesis, University of Aarhus, Denmark, 1997.
10. J. Palsberg. A provably correct compiler generator. In *Proceedings of the 4th European Symposium on Programming (ESOP92).* LNCS, 1992.
11. K. Slonneger and B.L. Kurtz. *Formal Syntax and Semantics of Programming Languages.* Adisson Wesley Publishing Company, Inc., New York, NY, 1995.
12. D. Watt. *Programming Language Syntax and Semantics.* Prentice-Hall, Inc., Englewoods Cliffs, New Jersey 07632, 1991.

A Modular SOS for Action Notation
(Extended Abstract)

Peter D. Mosses[1,2]

[1] BRICS and Department of Computer Science,
University of Aarhus, Denmark
[2] Visiting SRI International and Stanford University, USA

Abstract. Modularity is an important pragmatic aspect of semantic descriptions: good modularity is needed to allow the reuse of existing descriptions when extending or changing the described language. In denotational semantics, the issue of modularity has received much attention, and appropriate abstractions have been introduced, so that definitions of semantic functions may be independent of the details of how computations are modelled. In structural operational semantics (SOS), however, this issue has largely been neglected, and SOS descriptions of programming languages typically exhibit rather poor modularity; the original SOS given for Action Notation (the notation for the semantic entities used in action semantics) suffered from the same problem.
This extended abstract recalls a recent proposal, called MSOS, for obtaining a high degree of modularity in SOS, and introduces the MSOS description of Action Notation (which is provided only in the full paper). Due to its modularity, the MSOS description pin-points some complications in the design of Action Notation, and should facilitate the design of an improved version of the notation. It also provides a major example of the applicability of the MSOS framework.
The reader is assumed to be familiar with conventional SOS and with the basic concepts and constructs of Action Notation. The description of Action Notation is formulated entirely in Casl, the common algebraic specification language.

1 Background

This section recalls the main features of MSOS [9], Casl [3], and Action Notation [6]. Subsequent sections introduce and discuss the MSOS of Action Notation, which is provided in the full paper [10].

1.1 Modular SOS

Conventional SOS [1, 12] involves abstract syntax, computed values, configurations (some of which may be distinguished as terminal), and inference rules for (labelled) transitions. An SOS specifies a labelled transition system $(\Gamma, T, \mathbb{A}, \rightarrow)$, where Γ is the set of *configurations*, $T \subseteq \Gamma$ is the set of *terminal configurations*,

$\mathbb{A}$ is the set of *labels*, and $\rightarrow \subseteq \Gamma \times \mathbb{A} \times \Gamma$ is the *transition relation.* For configurations $\gamma, \gamma' \in \Gamma$ and labels $\alpha \in \mathbb{A}$, the assertion that $(\gamma, \alpha, \gamma')$ is in the transition relation is written $\gamma \xrightarrow{\alpha} \gamma'$.

Modular SOS, abbreviated MSOS [9], is a particularly simple and uniform discipline of SOS with the following features:

- Configurations $\gamma \in \Gamma$ are restricted to abstract syntax trees (where nodes may be replaced by the values that they have computed, as in conventional SOS).
- Initial configurations are pure syntax, and terminal configurations are simply computed values.
- All the usual semantic components of configurations (such as environments and stores) are incorporated in the labels $\alpha \in \mathbb{A}$ on transitions.
- The labels on transitions are equipped with a partial composition operation, written $\alpha \,\mathbf{;}\, \alpha'$ (associative whenever the composition is defined), and each label can always be composed on the left and right with identity labels $\iota \in \mathbb{I}[\mathbb{A}]$. The labels $\alpha \in \mathbb{A}$ are considered to be the arrows of a category, also written $\mathbb{A}$. The objects $o \in \mathbb{O}[\mathbb{A}]$ of the category correspond to the usual semantic components of configurations; let us refer to them as *states.*
- Transitions $\gamma_1 \xrightarrow{\alpha_1} \gamma'_1$ and $\gamma_2 \xrightarrow{\alpha_2} \gamma'_2$ may be adjacent in a computation only when $\gamma'_1 = \gamma_2$ and moreover the composition $\alpha_1 ; \alpha_2$ of their labels is defined.
- The actual representation of the labels α is abstracted from the rules that define the transition relations, allowing the former to be changed without invalidating the latter.

1.2 Label Categories

Label categories are defined succinctly using three standard label transformers, which correspond to some simple monad transformers. The following three label transformers, enriching label categories with further labels and states, are fundamental:

- Context_Info adds an extra component of a particular sort both to labels and to states, and its value is preserved by the *pre* and *post* operations. The composition $\alpha ; \alpha'$ is defined only when the new component has the same value in both α and α', and the composition preserves that value. This transformer is typically used for dealing with environments.
- Mutable_Info adds an extra component to states, and a *pair* of extra components (of the same sort) to labels, corresponding to the components of their *pre* and *post* states. The composition $\alpha ; \alpha'$ is defined only when this component has the same value in both *post*(α) and *pre*(α'). This transformer is typically used for dealing with stores.
- Emitted_Info adds an extra component only to labels. The composition $\alpha ; \alpha'$ combines the values of this component in α and α' using the operations of a given monoid. This transformer is typically used for dealing with output, the given monoid then being sequences with their concatenation.

The notation associated with the above label transformers is specified generically in CASL in [10]. It includes the operations *set*, for initializing or overwriting a particular component of a label or state, and *get*, for returning the value of a particular component (or a default value, if that component has not been set). Also the operations *get_pre* and *set_post* are provided in the case of MUTABLE_INFO, to avoid having to deal with pairs explicitly.

1.3 CASL Specifications

For defining abstract syntax, values, configurations, the notation used for labels, and transition relations, it is convenient to use CASL, the Common Algebraic Specification Language [3, 8]. CASL is quite expressive, providing direct support for specifying sort inclusions, partial operations, predicates, definedness assertions, and first-order axioms. CASL also provides datatype declarations (resembling grammars in BNF) that allow sorts equipped with constructors and selectors to be specified concisely. For structuring specifications, CASL provides union, extension, free extension (with initiality as a special case) and generic specifications. CASL does not allow the specification of inference rules for transitions, but we may write SOS transition rules as implications in CASL; the least relation satisfying the implications is obtained by letting the specification of transitions be a free extension.

Action Notation incorporates Data Notation [6, App. E], which provides various familiar datatypes: truth-values, numbers, characters, strings, lists, trees, sets, and finite maps, as well as some that are more closely connected with actions: data tuples, bindings, tokens, stores, cells, and agents. Data Notation is specified algebraically in the framework of Unified Algebras [4, 5]. Action Notation does not depend on the way that data is specified, except that a few primitive actions and yielders do require *sorts* of data as arguments (e.g., the action written 'choose natural' gives an arbitrary element of the sort natural), which is not allowed by CASL . To specify Data Notation in CASL, sorts that are to be used as arguments have to be represented by ordinary constants (or terms).

In fact the unified algebra treatment of sorts as values in a universe *Univ* can easily be simulated in CASL by distinguishing a subsort of 'individual' values $Indiv < Univ$, and declaring suitable operations and relations on *Univ*. The constant $nothing : Univ$ corresponds to an empty subsort of *Univ*. The unified algebra operations of sort union $\_\_|\_\_$ and intersection $\_\_\&\_\_$ are provided as ordinary operations on *Univ*, whereas the unified algebra subsort inclusion $\_\_ =< \_\_$ and individual inclusion $\_\_ :< \_\_$[1] are simply binary predicates in CASL. The predicate $u :< s$ holds iff the value u is both in *Indiv* and in the subsort represented by the value s. For instance, the unified algebra sort data is represented in CASL by declaring the subsorts $Data < Indiv$ and $DataSort < Univ$, and the constant $data : DataSort$, with $d : Data \iff d :< data$. The full properties of the general unified algebra notation are specified in CASL in [10].

[1] The unified algebra notation '$\_\_ : \_\_$' cannot be declared as a symbol in CASL.

Furthermore, Casl specifications of various basic abstract datatypes have recently been proposed [13], subsuming much of the standard Data Notation.

Therefore we may employ Casl for specifying both Action Notation (operationally, in the MSOS style) and Data Notation (algebraically), and avoid any involvement of the Unified Algebras framework in the foundations of Action Semantics.

1.4 Action Notation

Action Notation is a rich algebraic notation for expressing actions, which are used (along with data, and 'yielders' of data) to represent the semantics of constructs of conventional programming languages. Actions are essentially dynamic, computational entities. The performance of an action directly represents information processing behaviour and reflects the gradual, step-wise nature of computation: each step of an action performance may access and/or change the current information. Yielders occurring in actions may access, but not change, the current information. The evaluation of a yielder always results in a data entity (including a special entity used to represent undefinedness). For example, a yielder might always evaluate to the datum currently stored in a particular cell, which could change during the performance of an action, and become undefined when the cell is freed.

A performance of an action either: *completes*, corresponding to normal termination; or *escapes*, corresponding to exceptional termination; or *fails*, corresponding to abandoning an alternative; or *diverges*.

Action notation consists of several rather independent parts, corresponding to the following so-called 'facets' of information processing:

Basic: for specifying the flow of control in actions;
Functional: for specifying the flow of the data that are given to and by actions;
Declarative: for specifying the scopes of the bindings that are received and produced by actions;
Reflective: for specifying procedural abstraction and application;
Imperative: for specifying the allocation of storage for the values of variables; and
Communicative: for specifying (asynchronous) message passing.

Compound actions are formed from *primitive actions* and action *combinators*. Each primitive action is single-faceted, affecting information in only one facet—although any yielders that it contains may refer to all kinds of information. An action combinator determines control and information flow for each facet of the combined actions, allowing the expression of multi-faceted actions, such as an action that both (imperatively) reserves a cell of storage and then (functionally) gives the identity of the reserved cell. For instance, one combinator determines left-to-right sequencing together with left-to-right transient data flow, but letting the received bindings flow to its sub-actions; another combinator differs from that only regarding data flow: it concatenates any transients that the sub-actions give when completing, not passing transients between the actions at all. Some selections of control and information flow are disallowed, e.g., interleaving together with transient data flow between the interleaved sub-actions. In particular, imperative and communicative information processing always follows the flow of control.

Further informal explanation of the design of Action Notation may be found in the main sources for action semantics [6, 7, 14].

2 Introduction to the MSOS of Action Notation

The intended interpretation of Action Notation was originally defined [6, App. C] using a rather unorthodox style of SOS, exploiting the novel algebraic specification framework of Unified Algebras [4, 5]. The main features of unified algebras are that operations can be applied to, and return, entire sorts, and that individual values are regarded as singleton sorts. Transition relations can thus be represented as functions that map individual configurations to entire *sorts* of configurations (representing the sets of alternative transitions).

Unfortunately, the unorthodox style of the original SOS of Action Notation, combined with the unfamiliarity of Unified Algebras, made the specification somewhat inaccessible. Its lack of modularity also meant that even minor changes to Action Notation (or extensions of it, such as the proposal to allow agents to share storage [11]) might require a major reformulation of the given SOS. Moreover, to decrease the size of the description, the full Action Notation was reduced to a substantially-smaller kernel notation (by means of algebraic equations), and only the latter was given a direct operational semantics.

The full version of this paper gives an MSOS for all of Action Notation. It is structured in much the same way as [6, Apps. B and D], describing the various facets of Action Notation in turn; however, the semantics of each construct is here specified directly, without resort to an intermediate kernel notation.

Each section of the MSOS specifies the data notation, abstract syntax, computed values, configurations, label notation, and transition rules for the action notation in the facet concerned. The following explanatory comments apply to all the sections.

2.1 Data

Data notation is specified by reusing abstract datatypes that are already available, perhaps with renaming or instantiation of generic specifications and adding declarations and axioms for new notation. For instance:

spec BASIC_DATA =
  TRUTH_VALUES
    with *Truth_Value*, *true_value*, *false_value*, *either*
  and
    sorts *Data* < *Indiv*; *DataSort* < *Univ*

The symbols listed above after '**with**' are assumed to be declared by the CASL specification of TRUTH_VALUES (which uses slightly different identifiers than those in [6, App. E], to avoid confusion with the reserved CASL predicate symbols *true* and *false*). Many of the symbols of Data Notation are not valid CASL symbols, but generally become so once internal spaces and hyphens have been replaced by underscores.

As mentioned earlier, it is envisaged that the standard Data Notation used in Action Semantics may be replaced by a library of CASL specifications, perhaps incorporating the basic CASL datatype specifications that have recently been proposed [13].

By the way, only the data notation actually needed for the MSOS of Action Notation is specified in [10]. In particular, the declarations of constants such as $data : DataSort$, representing proper sorts in unified algebras, are omitted, since assertions such as $d :< data$ can be expressed equivalently as $d \in Data$, and $ds =< data$ as $ds \in DataSort$.

2.2 Syntax

Abstract syntax is specified in CASL using a datatype declaration, which resembles a BNF-like grammar. Mixfix notation is allowed—for instance, the following fragment specifies ***and*** as an infix operation:

spec BASIC_SYNTAX =
  BASIC_DATA **then**
    types $Action ::= \ldots \mid$ __***and***__$(Action; Action) \mid \ldots$;
          $Yielder ::= \ldots \mid$ *sort DataSort* $\mid \ldots$

The abstract syntax for actions and yielders extends the associated data notation, and data components are regarded as already evaluated.

It is possible to specify a syntactic congruence by adding axioms to the given datatype declarations, for instance asserting that A_1 ***and*** $A_2 = A_2$ ***and*** A_1, thereby reducing the need for various symmetric pairs of inference rules when specifying the transition relation.

By the way, several of the words used in Action Notation, such as '*and*', are reserved keywords in CASL, and cannot be complete tokens in CASL input symbols. So-called display annotations (not shown here) allow them to be produced in the formatted specification (using a distinct font, as in '***and***', to avoid confusion between symbols and keywords).

One might expect the types for the abstract syntax of actions and yielders for each facet of Action Notation to be specified as '**free**', to ensure that there can be no syntactic 'junk' (i.e., all syntactic values can be expressed by the declared constructors) nor 'confusion' (i.e., different terms denote different syntactic values, up to syntactic congruence) in models of the specification. However, that would prevent the subsequent combination of facets (as well as the extension of abstract syntax to configurations, see below). Instead, a free extension is specified *after* the facets have been combined.

2.3 Outcomes

The values that may be computed by action performance (and yielder evaluation) are specified algebraically in CASL, by declaring sorts, operations, and predicates, and asserting their essential properties. The specifications often use datatype declarations for conciseness. For instance:

```
spec FUNCTIONAL_OUTCOMES =
  BASIC_OUTCOMES and
  FUNCTIONAL_DATA
  then
    types  Terminated ::= sort Completed | ... ;
           Completed  ::= completed | gave(Data)
    axioms
    %[1]                gave(none) = completed;
    ...
```

2.4 Configurations

The 'value-added' syntax used for configurations is specified simply by adding further alternatives for the datatype declarations which specified abstract syntax: for each sort of the abstract syntax, the sort of value computed by elements of that sort is included as a subsort. Auxiliary syntactic constructs for use in configurations may be added here too.

In fact the configurations for non-distributed action performance are always the same, as specified by:

```
spec BASIC_CONFIGURATIONS =
  BASIC_SYNTAX and
  BASIC_OUTCOMES
  then
    type  Action ::= sort Terminated | __ @ __(Action; Action)
```

The sort *Terminated* (of values computed by actions) depends on the facet. (The auxiliary construct A_1 @ A_2 is used only in the basic facet, in connection with unfolding.)

The distributed performance of communicative actions by separate agents is described by embedding *Action* in an auxiliary sort of configurations, *Processing*, which allows collections of agents (with their actions), pending messages, and contracts all to be composed in parallel.

The datatype declaration for *Action* above augments the constructors for this sort, which is left loosely specified in BASIC_SYNTAX.

2.5 Labels

Each facet of Action Notation generally requires the transformation of the category of labels $\mathbb{A}$ to include one or more further components. This is specified concisely in CASL by instantiating one of the generic specifications corresponding to the three fundamental kinds of enrichment described in Section 1.2. For example, the functional facet specifies:

spec FUNCTIONAL_LABELS =
  BASIC_LABELS **and**
  FUNCTIONAL_DATA
  then
  CONTEXT_INFO
    [**sort** $\mathbb{A}$] [**op** *data* : *Index*]
    [**sort** *Data* < *ContextInfo* **op** *none* : *Data*]

which defines the operation $set(\alpha, data, d)$ to return a label α' with *data* component d, and the operation $get(\alpha, data)$ to return the *data* component of d, if defined (otherwise *none*).[2] The values of sort *Index* (such as *data*) may be thought of as selection indices; their only property is that different constants denote distinct values.

The fitting morphisms from the parameter specifications of CONTEXT_INFO to the argument specifications above are uniquely determined, and may therefore be left implicit.

2.6 Transitions

Transition rules are of three main kinds:

– Rules that allow performance of a compound construct to start (or continue) with a particular sub-construct: a transition for the sub-construct gives rise to a transition for the enclosing construct, often with the same unrestricted label α. For instance, the following rules allow interleaved performance of A_1 ***and*** A_2:

$$\%\% \frac{A_1 \xrightarrow{\alpha} A_1'}{A_1 \textbf{\textit{ and }} A_2 \xrightarrow{\alpha} A_1' \textbf{\textit{ and }} A_2;} \%\% \Rightarrow$$

$$\%\% \frac{A_2 \xrightarrow{\alpha} A_2'}{A_1 \textbf{\textit{ and }} A_2 \xrightarrow{\alpha} A_1 \textbf{\textit{ and }} A_2'} \%\% \Rightarrow$$

(The line between the conditions and the conclusion is not part of CASL notation, and has to be enclosed in comment signs '%%'.) Rules that specify the computation of a value by an atomic construct: the label on the transition is generally well-determined by the current state. For instance, the following rule lets the value computed by ***regive*** depend on the current state, which is not changed by the identity ι:

$$\%\% \frac{d = get(\iota, data)}{\textbf{\textit{regive}} \xrightarrow{\iota} gave(d)} \%\% \Rightarrow$$

[2] $set(\alpha, data, d)$ might be written even more suggestively as $\alpha[data := d]$, and $get(\alpha, data)$ as $\alpha.data$.

– Rules that reduce a compound configuration: once one or more components of a compound construct have computed values, the construct may be 'silently' reduced to a single computed value or syntactic component, the label on the transition being an identity ι. For instance, the following rule combines the values computed by performing the sub-actions of A_1 ***and*** A_2:

$$\mathit{gave}(d_1)\ \boldsymbol{and}\ \mathit{gave}(d_2) \xrightarrow{\iota} \mathit{gave}(\mathit{concatentation}(d_1, d_2))$$

An action is regarded as 'incorrect' when its performance can get stuck, i.e., lead to a configuration (other than a computed value) from which there is no further transition. For example, the action '***check abstraction_of*** A' is incorrect, since transitions are possible for '***check*** tv' only when $tv \in \mathit{Truth\_Value}$. The question of whether or not an arbitrary action is 'correct' is undecidable; a static semantics using type inference for action notation could however provide a useful decidable safe approximation to this notion.

The mathematical nature of the evaluation of yielders to data (sorts or individuals) is reflected by the labels on the transitions always being identities ι:

$$Y \xrightarrow{\iota} ds.$$

In general, the evaluation of yielders in a primitive action may be done in any order, and the result is independent of the chosen order. (Primitive actions are supposed to be indivisible, so a small-step gradual evaluation of yielder arguments would be incorrect.)

The ordinary transitive closure $\xrightarrow{\alpha}^{+}$ of $\xrightarrow{\alpha}$ is used in the rule for indivisible actions; its inductive definition is standard:

$$\%\% \frac{A \xrightarrow{\alpha} A'}{A \xrightarrow{\alpha}^{+} A'} \%\% \Rightarrow$$

$$\%\% \frac{A \xrightarrow{\alpha'} A' \wedge A' \xrightarrow{\alpha''}^{+} A'' \wedge \alpha = \alpha'; \alpha''}{A \xrightarrow{\alpha}^{+} A''} \%\% \Rightarrow$$

It is occasionally convenient to abbreviate two rules with the same conclusion by use of a single rule that has a disjunction of conditions. (Casl requires the intended grouping of a mixture of conjunctions and disjunctions to be made explicit, so there can be no doubt about the expansion of such an abbreviated rule.)

3 Discussion

The full MSOS of Action Notation is about 25 pages long, which is roughly twice as long as the original SOS for the kernel of Action Notation. The main reason for this expansion is not so much the difference in size between the kernel and full Action Notation, but more that the author went to great pains to achieve brevity in the original SOS. For instance, various subsorts that corresponded to restrictions of the original grammar were used—such subsorts are easy to express with the sort union operation of unified algebras. Auxiliary operations, effecting internal simplifications of the configuration, were introduced. Each combinator was classified into subsorts, e.g., according to whether it was sequential or interleaving; this allowed transitions to be specified for many combinators at once, rather concisely. Although such techniques might also be applicable in the MSOS of Action Notation, they would tend to undermine its modularity, and make it more difficult to cut down the description when removing entire facets.

The main hope for reducing the size of the MSOS of Action Notation is by means of a substantial simplification of Action Notation during the current reconsideration of its design. For instance, it appears that there is not much use for actions that simultaneously give some transient data and produce some bindings; eliminating them would allow all the hybrid combinators to be removed, and reduce the size of the MSOS of Action Notation by about 10%. The high degree of modularity of MSOS facilitates pin-pointing just which Action Notation constructs are excessively complicated.

It is hoped that the MSOS of Action Notation is much easier to follow than the original SOS—once one has grasped how dependencies between labels determine the flow of processed information, that is. (Readers who have difficulty with this aspect of MSOS might like to contemplate the reduction of MSOS to SOS [9] by moving the *pre* and *post* components of the labels to the configurations.)

Given the good modularity properties of MSOS, one might ask which is better: to describe the operational semantics of a programming language directly, using MSOS, or indirectly, using Action Semantics? In the author's opinion, it is generally better to use Action Semantics, for the following reasons.

The main advantage of the Action Semantics approach over MSOS is that the combinators of Action Notation provide concise abbreviations for particular *patterns* of MSOS (or SOS) transition rules. For instance, the combinator for sequential action performance without data-flow (written A_1 ***and_then*** A_2) abbreviates the pattern of transitions that occurs in many (M)SOS rules for left-to-right evaluation. A further advantage would show up in connection with the description of ML-style exceptions: Action Notation provides the ***escape*** primitive for escaping from normal action performance (with a value), and the combinator A_1 ***trap*** A_2 for trapping such escapes; in (M)SOS, the propagation of the exception value through all the syntactic constructs—apart from the exception handler—has to be specified explicitly.

However, MSOS also has some advantages over Action Semantics. Perhaps the main one is that the only unfamiliar notation provided by MSOS is that for the label transformers, whereas the full standard Action Notation is quite rich, and becoming familiar with it requires a significant initial investment of effort. Another stems from the very generality of the full Action Notation: its equational theory is too weak to be of much practical use. With MSOS, one may be able to prove stronger properties, exploiting awareness of the exact patterns of transitions and configurations that can arise.

Finally, for practical large-scale use of semantic descriptions, tool support is just as crucial as good modularity. Various tools have already been developed for Action Semantics (see other papers in this volume), whereas implementation of tools for MSOS is only just starting.

Those who have grown attached to the expressiveness provided by the framework of Unified Algebras may regret the switch to the more orthodox algebraic specification language CASL; indeed, the author himself has somewhat mixed feelings about abandoning this major application of the Unified Algebras framework, despite the ease with which it can be simulated in CASL . However, the adoption of CASL should not only increase the accessibility of Action Notation (by removing the need to learn first about Unified Algebras), but also it should pave the way for future exploitation of CASL libraries of standard abstract datatypes, and of CASL-based interfaces to existing tools (such as theorem-provers), in connection with action-semantic descriptions. The author was in any case happy to discover that CASL, itself originally designed for algebraic specification and development of software, appears to be quite well-suited also as a meta-notation for MSOS.

Acknowledgements The author is supported by BRICS (Centre for Basic Research in Computer Science), established by the Danish National Research Foundation in collaboration with the Universities of Aarhus and Aalborg, Denmark; by an International Fellowship from SRI International; and by DARPA-ITO through NASA-Ames contract NAS2-98073.

References

1. E. Astesiano. Inductive and operational semantics. In E. J. Neuhold and M. Paul, editors, *Formal Description of Programming Concepts*, IFIP State-of-the-Art Report, pages 51–136. Springer-Verlag, 1991.
2. CoFI. The Common Framework Initiative for algebraic specification and development, electronic archives. Notes and Documents accessible by WWW[3] and FTP[4].
3. CoFI Language Design Task Group. CASL – The CoFI Algebraic Specification Language – Summary. Documents/CASL/Summary, in [2], Oct. 1998.
4. P. D. Mosses. Unified algebras and institutions. In *LICS'89, Proc. 4th Ann. Symp. on Logic in Computer Science*, pages 304–312. IEEE, 1989.
5. P. D. Mosses. Unified algebras and modules. In *POPL'89, Proc. 16th Ann. ACM Symp. on Principles of Programming Languages*, pages 329–343. ACM, 1989.
6. P. D. Mosses. *Action Semantics.* Number 26 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1992.
7. P. D. Mosses. Theory and practice of action semantics. In *MFCS '96, Proc. 21st Int. Symp. on Mathematical Foundations of Computer Science (Cracow, Poland, Sept. 1996)*, volume 1113 of *LNCS*, pages 37–61. Springer-Verlag, 1996.
8. P. D. Mosses. CASL: A guided tour of its design. In J. L. Fiadeiro, editor, *Recent Trends in Algebraic Development Techniques, Proceedings*, volume 1589 of *LNCS*. Springer-Verlag, 1999.

[3] http://www.brics.dk/Projects/CoFI
[4] ftp://ftp.brics.dk/Projects/CoFI

9. P. D. Mosses. Foundations of Modular SOS (extended abstract). In *MFCS'99, Proc. 24th Intl. Symp. on Mathematical Foundations of Computer Science, Szklarska Poreba, Poland*, to appear in *LNCS*. Springer-Verlag, 1999. The full version is to appear in the BRICS Report Series.
10. P. D. Mosses. A modular SOS for Action Notation. To appear in BRICS Report Series, BRICS, Dept. of Computer Science, Univ. of Aarhus, 1999.
11. P. D. Mosses and M. A. Musicante. An action semantics for ML concurrency primitives. In *FME'94, Proc. Formal Methods Europe: Symposium on Industrial Benefit of Formal Methods, Barcelona*, volume 873 of *LNCS*, pages 461–479. Springer-Verlag, 1994.
12. G. D. Plotkin. A structural approach to operational semantics. Lecture Notes DAIMI FN–19, Dept. of Computer Science, Univ. of Aarhus, 1981.
13. M. Roggenbach and T. Mossakowski. Basic datatypes in CASL. Note M-6, in [2], Mar. 1999.
14. D. A. Watt. *Programming Language Syntax and Semantics*. Prentice-Hall, 1991.

The Abaco System
An Algebraic Based Action Compiler

Luis Carlos de Sousa Menezes [*]
Hermano Perrelli de Moura [**]

Federal University of Pernambuco - Department of Informatics
Caixa Postal 7851 - CEP 50732-970
Recife, Brazil

Abstract. In this article we propose an architecture for a new semantic directed compiler generator system named ABACO (Algebraic Based Action COmpiler). This system is based on a unified algebras compiler that produces C++ code from source specifications using an object oriented approach. The main advantage of this system is the ability of being easily extended with descriptions of new semantic entities defined as unified algebras specifications.

1 Introduction

In order to use action semantics descriptions for automatic compiler construction, many semantics directed compiler generation systems were proposed. Some examples of these systems are Actress [Mou93, Bro97], Oasis [Ørb93] and Cantor [Pal92]. These systems are limited in the sense that they use a fixed subset of action notation. This restriction difficultes the definition of new semantic entities, which could be useful to describe some particularity of the programming language.

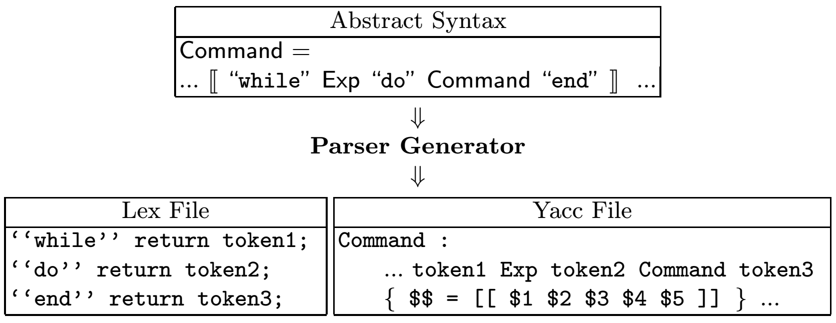
In order to solve that limitation, we present the architecture of a new action semantic based compiler generation system named ABACO (Algebraic Based Action COmpiler) [dSM98]. The most important characteristic of this system is to produce dynamic implementations for action semantics by processing the meta-notation used to describe the semantics entities of action notation. This property enables the system to be easily extended with the description of new semantic entities using the meta-notation.

The ABACO system is formed by the following tools: a *parser generator*; an *unified algebras compiler*; and an *action compiler*. The next sections will describe in more detail each one of these tools and the process of building a compiler from an action semantics description of a programming language.

[*] E-mail: `lcsm@di.ufpe.br`
[**] E-mail: `hermano@di.ufpe.br`

2 Parser Generator

The *parser generator* is the simplest system's tool. It just extracts the syntax definition and the lexical symbols from the language's abstract syntax and produces a lexical and syntactical descriptions that are compiled by tools like Lex [LS79] and Yacc [Joh79]. The resulting program is able to recognize source programs and call the functions produced by the system's tools to generate the action-program and the equivalent C++ program. An example of using the parser generator is showed in Figure 1.



```
''while'' return token1;
''do'' return token2;
''end'' return token3;
```

```
Command :
    ... token1 Exp token2 Command token3
    { $$ = [[ $1 $2 $3 $4 $5 ]] } ...
```

Fig. 1. Example of parser generation

3 Unified Algebras Compiler

The *unified algebras compiler* produces object oriented libraries from unified algebraic specifications [Mos88], meta-notation used by action semantics to describe its notation. These libraries have the following features:

- represent the terms of the specification as object-oriented expressions. For example, the term of the natural numbers specification (Figure 2): `sum 0 0`, will be expressed like the C++ object-oriented expression: `zero->sum(zero)`;
- when these expressions are executed, the returning object will correspond to the most simple equivalent term according the rewriting equations existing in the source specification. For example, when the expression: `zero->sum(zero)`, is executed the result value will be the object `zero`.

The process of building this library is based on the following similarities existing between the concepts of unified algebras and object-oriented paradigm:

- due the ambiguous nature of the unified algebras' sorts, they are equivalent to objects and classes of the object oriented paradigm. Classes implement sorts' properties like sort inclusion, operators, etc; and objects are used to represent them in object-oriented expressions;

– the sort inclusion relationship has similar properties of the class inheritance; and
– sorts' operators are represented by "operator" classes that represent the terms produced by the operator's application. Objects of these classes are produced by "operator" methods that implement the process of building a term using these operator. These operator's methods can:
  - return the result of another operator method's call, if there is a rewriting rule that matches with the represented term.
  - return a new instance of the equivalent operator class, otherwise.

Using these similarities we define a compilation process of unified algebras formed by the following steps: (1) simplification of the source specification, (2) source specification's analysis, (3) construction of the class structure and (4) definition of the generated methods. These steps will be defined in the next sections. To exemplify the process of unified algebras compilation we will use a natural numbers specification, showed in Figure 2.

introduces: natural, 0, successor _, sum _ _, fib _.
- 0 : natural.
- successor _ :: natural $\rightarrow$ natural.
- sum _ _ :: natural, natural $\rightarrow$ natural.

(1) sum 0 $x = x$.
(2) sum successor x y = successor sum x y.
- fib _ :: natural $\rightarrow$ natural.

(3) fib 0 = successor 0.
(4) fib successor 0 = successor 0.
(5) fib successor successor x = sum (fib successor x) (fib x).

Fig. 2. Natural numbers specification

3.1 Simplification of the Source Specification

The first step of the compilation is intended to remove some "syntactic sugar" existing in the model by replacing them to equivalent's forms. The main advantage of this simplification is to reduce the number of kinds of sentences to be handled in the next steps. Some examples of rules used to simplify unified algebras specifications are:

1. Functionality declarations:
   $o :: x(a_1, ...a_n),\ p_2, ... \rightarrow p_m.$

which declares that the operator o receives arguments formed by the application of another operator x can be simplified by declaring a new sort s equal to the term $x(a_1, ...a_n)$ and replacing it by s in the functionality declaration, resulting in these declarations:

   $o :: s,\ p_2,... \rightarrow p_m.$
   $s = x(a_1, ...a_n).$

2. Supersorts declarations like:

   $x \geq y.$

   can be expressed using equality and the join operation resulting in the following declaration:

   $x = x\ \mid\ y.$

3.2 Source Specification's Analysis

This step is intended to identify:

– Equality, individual and inclusion relationship of sorts defined by the source specification.
– The sorts produced by the operator's application.

These information can be found using rules obtained from the semantic of unified algebras. Some examples of these rules are:

1. Sentences like:

   $s_1 = ...\ \mid\ s_2\ \mid\ ...$

   defines that the sort s_2 is a subsort of the sort s_1.
2. Sentences like:

   $s_1 = ...\ \mid\ o(a_1,a_2...)\ \mid\ ...$

   defines that the operator o when applied with arguments of sorts a_1, a_2, etc; produces a subsort of the sort s_1.

In the example showed in Figure 2 the following information is obtained:

– 0 is an individual of sort natural
– the operators successor _, sum _ _ and fib _, produces subsorts of natural when applied to natural arguments.

3.3 Construction of the Class Structure

This step defines the classes and objects that will be produced by the process of unified algebras compilation. The definition of these elements uses the following rules, obtained from the similarities described before:

– The non-individual sorts will define classes that implement their properties.
– Each sort will have an object which represent it in the object-oriented world.
– If a sort r is subsort of another sort s then the class that implements the sort r is a subclass of the class that implements the sort s.

– If some operator o applied to arguments a_1, a_2,etc; produces individuals of sort s, then there will be defined an operator class to represent the application of o to subsorts of a_1, a_2,etc. This class will be subclass of the class generated by the sort s.

In the example of the natural numbers, we define the class structure showed (using a C++ like notation) in Figure 3. In this figure the name `Fib[natural]` represents the class designed to model the objects produced by the application of operator fib to subsorts of natural as arguments.

```
class Natural {}

Natural *natural = new Natural;
Natural *zero = new Natural;

class Successor[natural] : public Natural
{
   Natural *a1;
};

class Sum[natural,natural] : public Natural
{
   Natural *a1,*a2;
};

class Fib[natural] : public Natural
{
   Natural *a1;
};
```

Fig. 3. Generated class structure for natural numbers

3.4 Definition of the Generated Methods

The last step of the compilation process will decorate the class structure defined in the last step with instance methods that will implement the source specification's operators. The process defines two kinds of methods: default methods and operator methods.

Default methods enables the generated objects to answer general questions about the represented term (inclusion relationship, operator matching, etc). Figure 4 shows default methods generated by the operator sum _ _. The method

Match_sum checks if the term modeled by the current object is formed by application of the operator sum _ _ and updates the method's parameters with the arguments values.

```
int Natural::
    Match_sum(Natural *p1, Natural *p2)
{
   return 0;
}

int Sum[natural,natural]::
    Match_sum(Natural **p1, Natural **p2)
{
   *p1 = a1;
   *p2 = a2;
   return 1;
}
```

Fig. 4. Default methods for operator sum _ _

Operator methods implement the application of the operators. The current object and the arguments received by the method represent the term's arguments and the method's returning object represents the resulting term in its most reduced form. An operator o of functionality:

$o :: s_1,\ s_2,\ \ldots \rightarrow s_n$

will produce the following operator method skeleton:

```
C_$s_n$ C_$s_1$::$o$(C_$s_2$, ..., C_$s_{n-1}$)
{
    Check Equations
    Create Objects
}
```

where:

- C_x is the class that implements the sort x.
- *Check Equations* is the code segment that checks if the current building term matches with rewriting equations in the form:
  $o(a_1,\ldots,a_{n-1}) = t$
- *create objects* is the code segment responsible to create a new object of the correct operator class if no rewriting equation could be applied.

Figure 5 shows an operator method generated to implement the operator sum _ _. The first conditional command (lines 5-6) tests if the first argument (current object `this`) is the object used to represent the sort 0 (object `zero`), returning the value of the second argument if it is true. This segment implements the rewriting equation:

sum 0 $x = x$.

The next command (lines 7-8) checks if the first argument is formed by the application of the operator successor by calling the default method `Match_successor`. If the test succeeds the temporary variable `t1` will be assigned with the argument of the successor term and the command will return the expression `(t1->sum(a2))->successor()` that represents the right term of the following rewriting equation (implemented by this code segment):

sum successor x y = successor sum x y.

Finally, the method produces, in lines (9-10), a new object of the `Sum[natural,natural]` class if no rewriting rule could be applied. This class is designed to implement terms produced by the application of operator sum to proper sorts of natural.

```
(1)   Natural *Natural
(2)            ::sum(Natural *a2)
(3)   {
(4)      Natural *t1;
(5)      if (this==zero)
(6)         return a2;
(7)      if (this->Match_successor(&t1))
(8)         return t1->sum(a2)->successor();
(9)      return new Sum[natural,natural]
(10)                  (this,a2);
(11)  }
```

Fig. 5. Operator method for operator sum _ _

3.5 Performance Tests

In order to analyze the results of the unified algebras compiler, we have process the natural numbers specification using a beta version of the unified algebras compiler (UAC) and three others systems: OBJ3 [GKK$^+$88], a‘ compiler for OBJ [Ham95] and Maude [Mau]. These tests were made in a Sun SparcStation running the Solaris operating system. The results of these tests are showed in Table 1.

	UAC	OBJ3	OBJ Compiler	Maude
fib(13)	0.02	1.18	0.40	0.01
fib(14)	0.03	1.70	0.71	0.02
fib(20)	1.13	*	15.31	0.98
fib(23)	5.51	*	*	4.91

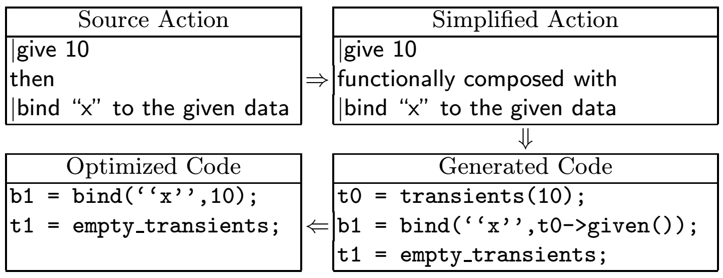
Table 1. Test of performance (in seconds, * means stack overflow)

4 Action Compiler

The *action compiler* produces C++ compilable code from program actions. The process of generating the C++ code is formed by the following steps:

1. The program action is converted into a more compact notation. This compact notation uses a more expressive action combinator to reduce the number of cases that the code generator should deal;
2. The action compiler produces a first version of the C++ compiled code from the simplified action produced in the previous step using some rules like the ones described in [Mou93]; and
3. The produced source code is optimized by reducing some existing redundancies, producing the final C++ code.

Figure 6 shows an example of action compilation.



```
|give 10
then
|bind "x" to the given data
```

```
|give 10
functionally composed with
|bind "x" to the given data
```

```
t0 = transients(10);
b1 = bind(``x'',t0->given());
t1 = empty_transients;
```

```
b1 = bind(``x'',10);
t1 = empty_transients;
```

Fig. 6. Example of action compilation

5 Building a Compiler Using the Abaco System

Building a compiler for a language $\mathcal{L}$ using the Abaco system requires the following steps, showed graphically in Figure 7:

1. The definition of the abstract syntax of $\mathcal{L}$ is processed by the *parser generator* which will generate a parser for $\mathcal{L}$.
2. The programming language description for $\mathcal{L}$ is processed by the *unified algebras compiler* that will produce a library, named *dynamic library.* The dynamic library will contain:
   – The abstract syntax tree of the $\mathcal{L}$ programs implemented as classes by compilation of the $\mathcal{L}$ abstract syntax;
   – Methods of the abstract syntax classes that will be able to give the meaning of programs using the action notation. This functionality is obtained by compiling the $\mathcal{L}$ semantic functions.
   – A class library that implements the action notation produced by compilation of the action notation description, imported by the programming language description
3. Compiled programs produced by the generated compiler will need a library which defines the data types used by the language to be correctly compiled. This library, named *static library*, is obtained compiling the semantics entities of $\mathcal{L}$ with the *unified algebras compiler.*
4. The dynamic library is linked with the *action compiler* to produce the code generator for the specified language. This program is able to produce C++ programs from AST produced by the parser generated in Step 1. The C++ programs generated by the code generator can be compiled using a generic C++ compiler.

The generated compiler is formed by the parser, the code generator, the static library and a generic C++ compiler (Figure 8).

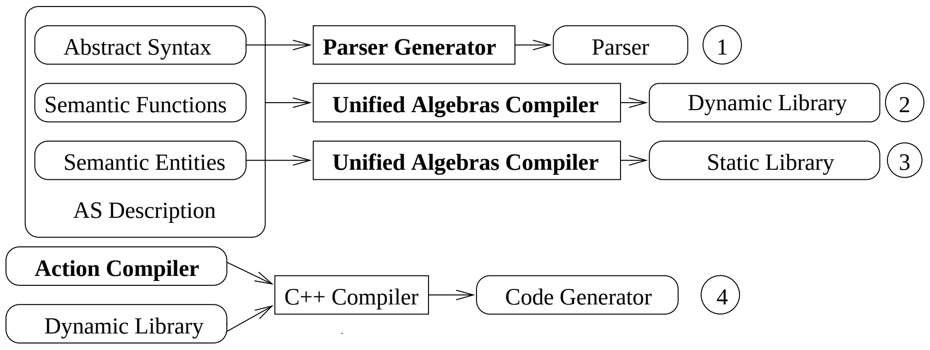


Fig. 7. Compiler generation using the ABACO System

6 Performance of the Abaco System

In order to test the system, an ABACO's implementation was created (without the described optimizations). This implementation was able to produce compilers for

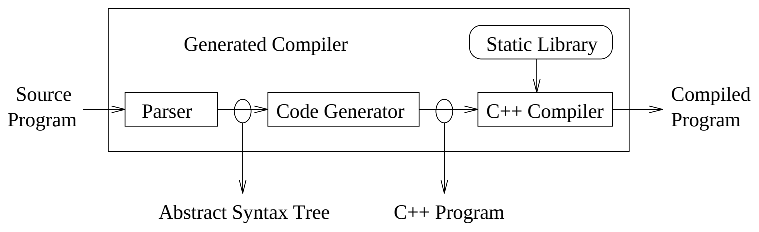


Fig. 8. Architecture of the generated compiler

small programming languages descriptions. We compared the execution of the programs in a small imperative pascal-like language, showed in figures 9 and 10 compiled using an ABACO generated compiler and their equivalents in C++ language compiled using the GNU C++ compiler. The results of the tests are showed in Table 2.

By analyzing the generated code we conclude that the ABACO system introduced a new performance problem formed by the implementation of basic types (numbers and truth-values) using classes, which introduced the overhead of method calls in the program's execution. This problem is not present in other compiler generation systems because they implements these types directly as primitive values. We are planing to work out a solution to implement these basic types directly as values and use object orientation to implement only the more complex ones.

```
program FIB =
let
   fun fib n =
       if (n==0 or n==1) then
          return 1;
       else
          return fib(n-1) + fib(n-2)
   end fun
in
   write(fib(25))
end program
```

Fig. 9. Test program for the ABACO system (FIB program)

```
program WHILE =
let
   var x;
   var y
in
   x := 10000;
   y := 0;
   while (x>0) do
     y := y + x;
     x := x - 1;
   end while
   write(y)
end program
```

Fig. 10. Test program for the Abaco system (WHILE program)

	GNU C++ compiler	Abaco compiler
FIB	0.015	4.510
WHILE	0.110	127.100

Table 2. Execution times (in seconds)

7 Conclusions and Future Works

By using the dynamic and static libraries to implement the semantic entities found in semantic descriptions of the source language, the implementation of action notation incorporates some language's properties and contains new semantic entities defined in the description. It makes the compilers produced by the Abaco system more flexible than the compilers produced by others compiler generator systems. The main drawback of using this system is the overhead of building the action notation implementation when processing the source description. We are working in faster analysis algorithms to reduce this overhead.

Future works on the Abaco system includes:

- Reimplementation of the system in Java to produce more portable compilers.
- Description of the compiler generation process using a formal language and verification of some properties like the formal soundness of the method. In the present moment we have only an informal verification of the method's soundness.
- Adapting to the Abaco's context several optimizations techniques proposed in research papers about automatic compiler generation. Some examples of these techniques can be found in [Mou93], [Bro97] and [Ørb93].

– Inclusion of some ABACO's specific optimization techniques like the implementation of basic types (numbers and booleans) as privitive values.

More information about the ABACO system and its beta implementation can be found in the RAT (Recife Action Tools) website [RAT].

References

[Bro97] Deryck F. Brown. *Sort Inference in Action Semantics.* PhD thesis, Department of Computing Science, University of Glasgow, 1997.

[dSM98] Luis Carlos de Sousa Menezes. Uso de orientação a objetos na prototipação de semântica de ações. Master's thesis, Universidade Federal de Pernambuco, 1998.

[GKK$^+$88] Joseph Goguen, Claude Kirchner, Hélèno Kirchner, Aristide Mégrelis, and José Meseguer. An introduction to OBJ3. In *Conference on Conditional Term Rewriting*, volume 308 of *Lecture Notes in Computer Science*, Springer 1988.

[Ham95] Lutz H. Hamel. *Behavioural Verification and Implementation of an Optimising Compiler for OBJ3.* PhD thesis, Oxford University Computing Laboratory, 1995.

[Joh79] S. C. Johnson. Yacc - yet another compiler compiler. Technical report, Bell Laboratories, 1979.

[LS79] M. Lesk and E. Schmidt. Lex - a lexical analyzer generator. Technical report, Bell Laboratories, 1979.

[Mau] Web site of maude. http://maude.csl.sri.com/.

[Mos88] P. D. Mosses. Unified algebras and action semantics. Departamental Report DAIMI PB–272, Aarhus University, Computer Science Department, Denmark, December 1988.

[Mou93] H. Moura. *Action Notation Transformations.* PhD thesis, University of Glasgow, Department of Computing Science, 1993.

[Ørb93] Peter Ørbæk. Analysis and optimization of actions. M.Sc. dissertation, Depto. of Computer Science, Univ. of Aarhus, September 1993.

[Pal92] Jens Palsberg. An automatically generated and provably correct compiler for a subset of Ada. In *ICCL'92, Proc. Fourth IEEE Int. Conf. on Computer Languages, Oakland*, pages 117–126. IEEE, 1992.

[RAT] Web site of the RAT Project. http://www.di.ufpe.br/∼rat.

This article was processed using the LaTeX macro package with LLNCS style

The Static and Dynamic Semantics of Standard ML

David A Watt

Department of Computing Science, University of Glasgow,
Glasgow G12 8QQ, Scotland. daw@dcs.gla.ac.uk

Abstract. This paper presents an action-semantic specification of the static and dynamic semantics of Standard ML. The specification is structured in the same way as the language itself, with separate modules for the core language and the full language.
Several aspects of the specification are of special interest. The specification of functors in the dynamic semantics turns out to be remarkably straightforward and transparent. The specification of polymorphic type inference in the static semantics uses sorts of types in a novel way, in order to make the specification truly declarative (as opposed to an encoding of the unification algorithm).
I will demonstrate that the action-semantic specification is not only more readable and more modular than the official natural-semantic specification [2]; it is actually more formal.

1 Introduction

Action semantics (AS) [4, 8] was conceived by Peter Mosses to be, above all, a *practical* formalism for specifying *real* programming languages. Until the 1990s, however, practical experience was somewhat limited. To the best of my knowledge, there was only one (nearly) complete AS specification of a real programming language, namely that of Standard Pascal [6].

Pascal is, of course, a classical imperative language. It is equally important to test the effectiveness of AS for specifying languages in other paradigms, including functional and object-oriented languages. For this purpose I chose to specify Standard ML (SML) and Modula-3.[1]

I had already written an AS specification of the dynamic semantics of ML using a now-obsolete version of action notation [7]. However, that work addressed only the ML bare language, which (as its name implies) is a small functional language, much simpler than SML.

SML is certainly a major challenge to any semantic formalism. It is a type-safe polymorphic functional programming language, with imperative features such as reference, assignment, and exceptions. It has a module layer that supports "structures" (simple modules), and "functors" (structures parameterised with

[1] My choice of Modula-3 just preceded the explosion of interest in Java. I later abandoned the Modula-3 specification in favour of Java [1].

respect to other structures). Its static semantics is characterised by polymorphic type inference.

The specific goals of this project were as follows:

1. To test the effectiveness of AS for specifying the dynamic semantics of SML.
2. To test the effectiveness of AS data notation for specifying the static semantics of SML.
3. To compare the AS specification of SML with the official natural-semantic (NS) specification of Milner *et al.* [2].

The rest of this paper is structured as follows. Section 2 is a brief overview of the SML language, focussing on those features that proved most challenging to specify. Section 3 is an overview of the AS specification, focussing on its modular structure. Section 4 presents selected extracts from the dynamic semantics of ML's core layer, and compares them with corresponding extracts from the NS specification. Section 5 similarly presents, discusses, and compares the static semantics of ML's core layer. Section 6 presents and discusses the dynamic semantics of ML's module layer. Section 7 concludes.

2 The Standard ML Language

SML is stratified into two layers:

– The *core layer* is a conventional polymorphic functional language that processes ordinary typed values. ML values include primitive values, tuples, constructions, references, arrays, recursively-structured values, functions, and exceptions. All of these are first-class values.
– The *module layer* adds structures, signatures, and functors. A *structure* is just an encapsulated group of component declarations. A *signature* defines the types of a structure's components but not their implementations.[2] A *functor* is a structure parameterised with respect to another structure. Since the parametric structure is unknown, the functor declaration must state its signature. (It may also, optionally, state the result structure's signature.)

The stratification of SML is rigorous. Functions are parameterised with respect to ordinary values only; functors are parameterised with respect to structures only. In other words, structures are *not* first-class.

In this paper, I use *Core ML* when referring to the language subset consisting of the core layer alone; *SML* when referring to the full language consisting of both the core layer and the module layer; and *ML* when neutrality is sufficient.

Fig. 1 illustrates some features of the core layer.

ML has numerous name spaces: value-variables, value-constructors, type-variables, type-constructors, exception-constructors, etc. Of these, only type-variables (`'a`, `'b`, etc.) are syntactically distinct. In the example of Fig. 1(a), `pair` is both a type-constructor and a value-constructor in the same scope. The

[2] A signature can be viewed as the "type" of a structure.

syntactic context allows the compiler to distinguish: in the declaration of `p`, the first occurrence of `pair` is a type-constructor, whereas the second occurrence of `pair` is a value-constructor. However, the same identifier cannot be both a value-variable and a value-constructor in the same scope. So a declaration of `pair` as a value-variable denoting a function would hide its declaration as a value-constructor in the example, since in the context of an application `pair(...)` the compiler could not otherwise distinguish between the value-variable and the value-constructor.

The syntax of an application is heavily overloaded in ML. It is used for both constructions and function calls. Moreover, ML makes no semantic distinction (and very little syntactic distinction) between operators and identifiers denoting functions. Thus the expression “`n + 1`” simply abbreviates the function call “`op +(n, 1)`”; and the expression “`m + 2 * n`” abbreviates the function call “`op +(m, op *(2, n))`”.[3]

ML’s imperative features are based mainly on reference types. In the example of Fig. 1(b), the expression “`ref 0`” allocates an integer cell and initialises it to zero. Here `ref` denotes the special *reference-value-constructor*. The declaration “`val count = ref 0`” binds `count` to a newly allocated and initialised integer cell. The expression “`count := !count + 1`” increments the integer contained in that cell. Here “`!`” denotes the special dereferencing function, and “`:=`” denotes the special *assigner* function. The above expression abbreviates `op :=(count, op +(!count, 1))`”.

Returning to Fig. 1(a), notice that the type of the function `snd` is not stated. ML relies on polymorphic type inference. Many types could be inferred for `snd`, including `'t pair -> 't`, `int pair -> int`, and `('u list)pair -> 'u list`. Of these, `'t pair -> 't` is the *principal type* — the most general type, which is unique up to renaming of type-variables. In type-theoretic notation, this principal type would be written as a *type scheme*:

$$\forall\tau.(\tau \text{ pair} \to \tau)$$

which emphasises that τ is universally quantified over the type expression.

Fig. 2 illustrates the module layer. The first declaration defines `ORDERED` to be a signature that is satisfied by any structure whose components include *at least*: (i) a type named `t`, and (ii) a function named `less` that takes a pair of `t` arguments and returns a `bool` result. The second declaration defines `OrderedString` to be one such structure, where the type named `t` happens to be `string`. The third declaration defines `Set` to be a functor that maps an argument structure to a result structure. Because the functor’s formal parameter is written “`Members: ORDERED`”, the argument structure is locally named `Members` and must satisfy the `ORDERED` signature. The functor maps `Members` to a result structure that in fact supports sets of values of type `Members.t`. The last declaration applies functor `Set` to structure `OrderedStrings`, resulting in a structure that supports sets of strings.

[3] “`op +`” is the prefix equivalent of the infix “`+`”. “`(..., ...)`” is just a tuple constructor.

(a) Value-variables, value-constructors, type-constructors, and polymorphism:

```
let
   datatype 'a pair = pair of 'a * 'a;
   fun snd (pair of (x, y)) = y;
   val p: string pair = pair("Mosses", "Watt")
in
   snd p
```

(b) Imperative features:

```
let
   val count = ref 0;
   fun inc () = count := !count + 1
in
   (...; inc(); ...; inc(); ...)
```

Fig. 1. Examples of code in the ML core layer.

```
signature ORDERED =
   sig
      type t;
      val less: t * t -> bool
   end

structure OrderedStrings: ORDERED =
   struct
      type t = string;
      fun less (x: string, y: string) = x < y
   end

functor Set (Members: ORDERED) =
   struct
      type set = set of (Members.t)list;
      (* A set is represented by an ordered list. *)
      val empty: set = set[];
      fun single (x: Members.t) = set[x];
      fun union (s1: set, s2: set) =
         ... if Members.less(x, y) then ...
   end

structure StringSets = Set(OrderedStrings)
```

Fig. 2. Example of code in the ML module layer.

3 Overview of the Action-Semantic Specification

To facilitate comparison of the AS specification of SML with the official NS specification [2] (goal 3 in the introduction), I chose to structure the AS specification in the same way as the NS specification. A (slightly simplified) overview of the structure is shown in Fig. 3.

The symmetry of the structure should be evident. Modules **Core ML** and **SML** specify Core ML and SML, respectively. Each has submodules **Abstract Syntax**, **Static Semantics**, **Dynamic Semantics**, and **Programs**, with the dependencies between them made explicit by the **needs** clauses. Moreover, the inclusion of Core ML in SML is made explicit by the **includes** clauses: in most cases, a submodule of **SML** includes the corresponding submodule of **Core ML**.

The submodules are in turn subdivided (not shown in Fig. 3). Each **Static Semantics** and **Dynamic Semantics** module is subdivided into **Semantic Functions** and **Semantic Entities**. Each **Abstract Syntax** and **Semantic Functions** module is subdivided into **Expressions**, **Declarations**, and so on.

Comparison

We see already an important difference between the NS and AS specifications of SML. The AS specification is formally structured into modules, with explicit dependencies and inclusions, using the meta-notation of [4]. In the NS specification [2], the "modules" are just book chapters, and the dependencies and inclusions are stated informally.

4 Core ML/Dynamic Semantics

Fig. 4 and 5 show extracts from the dynamic semantics of Core ML (value) declarations, patterns, and expressions. The denotations are actions with a variety of possible outcomes:

- binding is the normal outcome for a declaration or pattern;
- giving a value is the normal outcome for an expression;
- diverging is a possible outcome for an expression that might loop forever, or any other construct that evaluates such an expression;
- changing state is a possible outcome for an expression that might have side effects, or any other construct that evaluates such an expression;
- escaping with a packet is a possible outcome for an expression that might raise an exception, or any other construct that evaluates such an expression;
- misfitting is a possible outcome for a pattern that might be a bad fit for the given value.

The outcomes changing state and misfitting are defined in **Semantic Entities**.

Identifiers are mapped by the operations var _, con _, etc., into disjoint subsorts of token. When the value declaration `val red = 0` is elaborated, the *value-variable* var "red" is bound. When the data-type declaration `datatype colour`

Core ML.
Core ML/Abstract Syntax.

needs: [4]/**Data Notation**.

Core ML/Static Semantics.

needs: **Abstract Syntax**, [4]/**Data Notation**.

Core ML/Dynamic Semantics.

needs: **Abstract Syntax**, [4]/**Data Notation**, [4]/**Action Notation**.

Core ML/Static Basis.

needs: **Static Semantics**.

Core ML/Dynamic Basis.

needs: **Dynamic Semantics**.

Core ML/Programs.

needs: **Abstract Syntax**, **Static Semantics**, **Dynamic Semantics**,
Static Basis, **Dynamic Basis**,
[4]/**Data Notation**, [4]/**Action Notation**.

SML.

SML/Abstract Syntax.

needs: [4]/**Data Notation**.
includes: **Core ML/Abstract Syntax**.

SML/Static Semantics.

needs: **Abstract Syntax**, [4]/**Data Notation**.
includes: **Core ML/Static Semantics**.

SML/Dynamic Semantics.

needs: **Abstract Syntax**, [4]/**Data Notation**, [4]/**Action Notation**.
includes: **Core ML/Dynamic Semantics**.

SML/Programs.

needs: **Abstract Syntax**, **Static Semantics**, **Dynamic Semantics**,
Core ML/Static Basis, **Core ML/Dynamic Basis**,
[4]/**Data Notation**, [4]/**Action Notation**.

Fig. 3. Structure of AS specification of ML.

`= red | green | blue` is elaborated, on the other hand, the *value-constructors* con "red", con "green", and con "blue" are bound. (Each value-constructor is bound to itself.) In this way, elaborating a group of declarations produces a set of bindings for value-variables, value-constructors, etc., all mixed together.

In Fig. 5 the key equation is (4), which specifies the semantics of an application $[\![E_1\ E_2]\!]$. After evaluating E_1 and E_2, this resolves itself into five cases. If the value of E_1 is a normal function, that function (represented by an abstraction) is enacted as usual. If the value of E_1 is the special assigner function (denoted by "`:=`"), the value of E_2 will be a pair consisting of a reference (cell) and a value, so that value is stored in that reference. If the value of E_1 is a normal value-constructor, a construction is made of that value-constructor and the value of E_2. If the value of E_1 is the special reference-value-constructor (denoted by `ref`), a reference is allocated and the value of E_2 is stored in it. Finally, if the value of E_1 is an exception-constructor, an exception is made of the exception-constructor and the value of E_2.[4]

Comparison

The NS specification [2] handles separate name spaces for value-variables, value-constructors, etc., by carrying around a tuple of separate environments. The usual operations for retrieving a binding from an environment, adding a binding to an environment, etc., are lifted to the tuple of environments. The AS specification shows that this complexity is unnecessary.

In the AS specification it is the semantic equations that decide which of the operations var _, con _, etc., should be used to map a given identifier occurrence to a token. The NS specification, on the other hand, assumes an abstract syntax in which each identifier occurrence has already been mapped to a value-variable, a value-constructor, or whatever. Moreover, this mapping is defined only informally.

Fig. 6 shows a short extract from the NS specification. I have made cosmetic changes to the notation of [2] in order to facilitate comparison.

The inference rule in Fig. 6(a) specifies the dynamic semantics of the expression-series $[\![E_1\ \text{";"}\ E_2]\!]$. Compare this with equation (5) of Fig. 5. The styles are very different, as we would expect: AS is English-like and verbose, NS uses a logical notation that is mathematical and cryptic.

Unexpectedly, however, the AS specification is actually more formal! The "_ then _" combinator in equation (5) states precisely that evaluate E_1 is performed before evaluate E_2, with the received bindings distributed to both of these actions. The inference rule in Fig. 6(a) does specify the distribution of bindings (e), but is silent about the flow of control. In fact, it exploits the so-called *state convention*, whereby states (s) are omitted from most inference rules. The first

[4] In ML, evaluating an expression can yield an exception as its (normal) result. This is not the same as *raising* an exception, whereby evaluation is abandoned and the exception is propagated to a handler in some enclosing expression. Raising an exception is specified in the AS specification by an escape.

Core ML/Dynamic Semantics/Semantic Functions/Declarations.

introduces: establish _ .

```
  • establish _ :: Declaration →
          action [binding | escaping with a packet | diverging | changing state]
                 [using current bindings | current state] .

(1) establish ⟦ “val” $B$:Value-Binding ⟧ =
          establish $B$ .

  • establish _ :: Value-Binding →
          action [binding | escaping with a packet | diverging | changing state]
                 [using current bindings | current state] .

(2) establish ⟦ $P$:Pattern “=” $E$:Expression ⟧ =
          evaluate $E$ then
          | fit $P$
          | else misfittingly
          | escape with the packet of the bind-exception .

(3) establish ⟨ $B_1$:Value-Binding “and” $B_2$:Value-Binding ⟩ =
          establish $B_1$ and then establish $B_2$ .
```

Core ML/Dynamic Semantics/Semantic Functions/Patterns.

introduces: fit _ .

```
  • fit _ :: Pattern →
          action [completing | misfitting | binding]
                 [using the given value | current bindings | current storage] .

(1) fit “_” =
          complete .

(2) fit $V$:Value-Variable =
          bind var $V$ to the given value .
```

Fig. 4. AS specification of dynamic semantics of Core ML declarations and patterns (extracts).

Core ML/Dynamic Semantics/Semantic Functions/Expressions.

introduces: evaluate _ , serially evaluate _ .

```
• evaluate _ :: Expression →
        action [giving a value | escaping with a packet | diverging | changing state]
               [using current bindings | current state] .

(1) evaluate $V$:Value-Variable =
        give the value bound to var $V$ .

(2) evaluate $C$:Value-Constructor =
        give con $C$ .

(3) evaluate ⟦ "let" $D$:Declaration "in" $E$:Expression "end" ⟧ =
        furthermore establish $D$
        hence evaluate $E$ .

(4) evaluate ⟦ $E_1$:Expression $E_2$:Expression ⟧ =
        |evaluate $E_1$ and then evaluate $E_2$
        then
        ||check there is given (a function, a value) and then
        |||enact the application of the given function#1 to the given value#2
        |or
        ||check there is given (the assigner, a pair) and then
        |||give the components of the given pair#2 then
        ||||store the given value#2 in the given reference#1 and
        ||||give the unit-record
        |or
        ||check there is given (a normal-value-constructor, a value) and then
        |||give the construction of them
        |or
        ||check there is given (a reference-value-constructor, a value) and then
        ||||allocate an reference and
        ||||give the given value#2
        |||then
        ||||store the given value#2 in the given reference#1 and
        ||||give the given reference#1
        |or
        ||check there is given (an exception-constructor, a value) and then
        |||give the exception of them .

• serially evaluate _ :: Expression-Series →
        action [giving a value | escaping with a packet | diverging | changing state]
               [using current bindings | current state] .

(5) serially evaluate ⟨ $E_1$:Expression ";" $E_2$:Expression ⟩ =
        evaluate $E_1$ then evaluate $E_2$ .
```

Fig. 5. AS specification of dynamic semantics of Core ML expressions (extracts).

Expressions $e \vdash E \Rightarrow v/p$

(a) Using the state and exception conventions:

$$\frac{e \vdash E_1 \Rightarrow v_1 \qquad e \vdash E_2 \Rightarrow v_2}{e \vdash [\![E_1 \text{“;”} E_2]\!] \Rightarrow v_2}$$

(b) In full:

$$\frac{e, s \vdash E_1 \Rightarrow v_1, s' \qquad e, s' \vdash E_2 \Rightarrow v_2, s''}{e, s \vdash [\![E_1 \text{“;”} E_2]\!] \Rightarrow v_2, s''}$$

$$\frac{e, s \vdash E_1 \Rightarrow p, s'}{e, s \vdash [\![E_1 \text{“;”} E_2]\!] \Rightarrow p, s'}$$

$$\frac{e, s \vdash E_1 \Rightarrow v_1, s' \qquad e, s' \vdash E_2 \Rightarrow p, s''}{e, s \vdash [\![E_1 \text{“;”} E_2]\!] \Rightarrow p, s''}$$

$(e \in \mathsf{Env}; s \in \mathsf{State}; v \in \mathsf{Val}; p \in \mathsf{Pack})$

Fig. 6. NS specification of dynamic semantics of Core ML expressions (extract).

inference rule in Fig. 6(b) shows the effect of re-instating the states. This expanded inference rule does specify the flow of control, indirectly, by means of the state thread.

But what happens if evaluation of E_1 raises an exception? In equation (5) of Fig. 5, this is again captured by the “_ then _” combinator: if evaluate E_1 escapes, the whole action escapes, i.e., the exception is propagated out of the expression-series. In the NS specification, if E_1 raises an exception, its result is an exception *packet* (p) rather than a value. However, the inference rule in Fig. 6(a) is silent about this issue too. In fact, it exploits the so-called *exception convention*, whereby exception propagation is omitted from most inference rules. The second inference rule in Fig. 6(b) re-instates exception propagation. It shows that, when E_1 evaluates to a packet p, E_2 is skipped and p is taken as the result of the whole expression-series. The third inference rule in Fig. 6(b) similarly shows what happens when E_2 raises an exception.

The state convention and exception convention are essential to keep the NS specification manageable. Without these conventions, there would be 2–3 times as many inference rules, and these would be less readable. Looking at the problem from a different viewpoint, if Milner *et al.* had chosen to develop their NS specification by building up from a purely applicative subset of ML, as soon as they added state and exceptions they would have had to modify most of their existing inference rules [9].

Despite their pragmatic importance, the state convention and exception convention are informal. A more principled approach to the same problem is possible, as shown by Peter Mosses' recent development of *modular SOS* [5].

5 Core ML/Static Semantics

We have already seen, in Section 2, that an ML expression does not, in general, have a unique type, although it does have a unique principal type. Any formal specification must adopt one of two alternative approaches:

1. Define a function that maps each expression to its principal type.
2. Define a relation between expressions and types.

Approach 1 makes the specification, in effect, an encoding of the unification algorithm.[5] Approach 2 is the one adopted in [2], the judgements of natural semantics being well suited for expressing relations.

In order to facilitate goal 3 (Sect. 1), I decided to adopt approach 2. At first sight this seems paradoxical, for the semantic functions of an AS specification are (what else?) functions! But the unified-algebraic foundations underlying AS can be exploited to define functions whose results are sorts, rather than individuals. Such functions model relations.

In the AS specification, the function "the types of _ in _" maps each expression to the *sort* of all its possible types. Similarly, the function "the elaboration of _ in _" maps each declaration to the *sort* of all environments[6] that it might possibly produce. Each of these functions has a second operand that is a context.[7]

A corollary of the decision to adopt approach 2 above is that the AS specification of the static semantics of ML does not use action notation. Actions cannot give sorts, only individual data.

In any case, it is surely preferable to specify static semantics in a declarative fashion; whereas dynamic semantics benefits from an operational specification in, for example, action notation.

Fig. 7 shows extracts from the static semantics of Core ML expressions. The key equation is (4), which specifies the typing of an application $[\![E_1\ E_2]\!]$. It simply asserts that if the types of E_1 include a function type whose domain is t' and whose range is t, and if the types of E_2 include t', then the types of the application include t.[8]

We see that the use of a semantic function mapping each expression to a sort of types, together with the use of notation for testing an individual type's membership of a sort of types, allows us to define the expression–type relation in data notation.

Comparison

Figs. 9 and 10 show inference rules for the static semantics of selected expressions and declarations in the NS specification [2].

[5] This approach has been successfully tested by Deryck Brown, in an unpublished AS specification of the static semantics of a small subset of ML.
[6] An *environment* binds value-variables, value-constructors, etc., to their types.
[7] A *context* consists of an environment plus other information of no concern here.
[8] Equation (4) in Fig. 7 does not need several cases like equation (4) in Fig. 5, because normal and special functions, normal and special value-constructors, and exception-constructors all have function types.

Core ML/Static Semantics/Semantic Functions/Expressions.

introduces: the types of _ in _ .

- the types of _ in _ :: Expression, context $\rightarrow$ type .
- the types of E:Expression in c:context $\leq$ type .

(1) the types of V:Value-Variable in c:context =
        the types generalised by the type-scheme bound to var V in c .

(2) the types of C:Value-Constructor in c:context =
        the types generalised by the type-scheme bound to con C in c .

(3) the elaboration of D in c :- e:environment ;
    the overlay of (e, c) with distinct type-names = c':context ;
    the types of E in c' :- t:type
    $\Rightarrow$
    the types of ⟦ "let" D:Declaration "in" E:Expression "end" ⟧ in c:context :- t .

(4) the types of E_1 in c :- the function-type of (t':type, t:type) ;
    the types of E_2 in c :- t'
    $\Rightarrow$
    the types of ⟦ E_1:Expression E_2:Expression ⟧ in c:context :- t .

Fig. 7. AS specification of static semantics of Core ML expressions (extract).

Core ML/Static Semantics/Semantic Functions/Declarations.

introduces: the elaboration of _ in _ .

- the elaboration of _ in _ :: Declaration, context $\rightarrow$ environment .
- the elaboration of D:Declaration in c:context $\leq$ environment .

(1) the elaboration of B in c :- e:variable-environment ;
    the imperative closure of e with respect to (
        the free type-variables of the environment of c,
        the value-variables expansively defined by B)
        = e':variable-environment ;
    the intersection of (the type-variables scoped at B, the free type-variables of e')
        = the empty-set
    $\Rightarrow$
    the elaboration of ⟦ "val" B:Value-Binding ⟧ in c:context :- e' .

Fig. 8. AS specification of static semantics of Core ML declarations (extract)

Expressions $\quad c \vdash E \Rightarrow t$

$$\frac{c \vdash V \succ t}{c \vdash V \Rightarrow t}$$

$$\frac{c \vdash C \succ t}{c \vdash C \Rightarrow t}$$

$$\frac{c \vdash D \Rightarrow e \qquad c \oplus e \vdash E \Rightarrow t}{c \vdash [\![\text{“let”}D\text{“in”}E\text{“end”}]\!] \Rightarrow t}$$

$$\frac{c \vdash E_1 \Rightarrow t' \to t \qquad c \vdash E_2 \Rightarrow t'}{c \vdash [\![E_1 E_2]\!] \Rightarrow t}$$

$(c \in \mathsf{Context}; t \in \mathsf{Type})$

Fig. 9. NS specification of static semantics of Core ML expressions (extracts).

Declarations $\quad c \vdash D \Rightarrow e$

$$\frac{c + u \vdash B \Rightarrow e \qquad e' = \mathrm{Clos}_{c,B}(e) \qquad u \cap \mathrm{tyvars}(e') = \emptyset}{c \vdash [\![\text{“val”}_u B]\!] \Rightarrow e' \text{ in } \mathsf{Env}}$$

$(c \in \mathsf{Context}; u \in \mathsf{TyVarSet}; e, e' \in \mathsf{VarEnv})$

Fig. 10. NS specification of static semantics of Core ML declarations (extract).

The AS specification of the static semantics follows very closely the structure of the NS specification. This can be seen by comparing Figs. 7 and 8 with Figs. 9 and 10, respectively. The main difference is in readability.

The NS specification relies on a variety of auxiliary operations. Some like "tyvars(e)" are intuitive and easy to internalise; others like "$\mathrm{Clos}_{c,B}(e)$" are very much less so. The AS specification has similar auxiliary operations,[9] such as "free type-variables of e" and "imperative closure of e with respect to $(\ldots, \ldots)$". The latter are certainly more readable and arguably easier to internalise.

More importantly, all auxiliary operations in the AS specification of the static semantics are formally specified in **Semantic Entities**. Some of the auxiliary operations in [2] are specified only informally or semi-formally.

6 SML/Dynamic Semantics

In the AS specification of the dynamic semantics of SML, a structure is represented by an encapsulated set of bindings. A functor is represented by an abstraction that uses a given structure (its argument) and gives a structure (its result). These representations are summarised in Fig. 11.

Figs. 12 and 13 show extracts from the AS specification of the dynamic semantics of SML structure-expressions and (structure and functor) declarations, respectively.

Equations (3) and (4) in Fig. 13 specify structure-bindings. Equation (4) shows the effect of stating a signature for the structure. The operation "the interface described by G" maps signature G to an interface (see Fig. 11). The operation "s restricted to i" prunes the structure s of any components not in the interface i.[10]

Of particular interest are equation (3) in Fig. 12, which specifies functor application, and equation (5) in Fig. 13, which specifies functor-bindings. These should look familiar to anyone accustomed to AS. They are in fact analogous to semantic equations for function application and function definition, with functors replacing functions, and structures replacing values.[11] Note, however, that the operation "_ restricted to _" is applied to the functor's argument structure. The argument might be a structure with more components than those represented in the parameteric structure's signature; these extra components are simply ignored by the functor.

[9] However, specifying the auxiliary operations formally in the AS specification exposed opportunities to simplify some of them.

[10] This operation is partial. The structure s must have *at least* the components represented in the interface i, and these must have the same types, otherwise the operation yields nothing.

[11] It is actually simpler to specify functors than functions in ML, since the semantics of function-bindings are complicated by recursion and clausal definitions.

SML/Dynamic Semantics/Semantic Entities.

introduces: structure , structure-bindings , structure of _ .

- structure = structure of structure-bindings .
- structure-bindings = map[structure-label to structure-field] .

introduces: interface , interface of _ , _ restricted to _ .

- interface = interface of (interface-map, value-variable-set, exception-constructor-set) .
- _ restricted to _ :: structure, interface → structure (*partial*) .

introduces: functor , functor of _ , application _ to _ .

- functor = functor of abstraction [giving a structure | escaping with a packet | diverging | changing state] [using the given structure | current state] .
- application _ to _ :: functor, structure → abstraction [giving a structure | escaping with a packet | diverging | changing state] [using current state] (*total*) .

Fig. 11. AS specification of dynamic semantic entities of SML (extracts).

SML/Dynamic Semantics/Semantic Functions/ Structure-Expressions.

introduces: structure-evaluate _ .

- structure-evaluate _ :: Structure-Expression → action [giving a structure | escaping with a packet | diverging | changing state] [using current bindings | current state] .

(1) structure-evaluate ⟦ "struct" D:Structure-Level-Declaration "end" ⟧ =
establish D hence
give the structure of the current bindings .

(2) structure-evaluate I:Long-Structure-Identifier =
give the structure denoted by I .

(3) structure-evaluate ⟦ I:Functor-Identifier "(" S:Structure-Expression ")" ⟧ =
structure-evaluate S then
enact the application of the functor denoted by I to the given structure .

Fig. 12. AS specification of dynamic semantics of SML structure-expressions.

SML/Dynamic Semantics/Semantic Functions/Declarations.

```
• establish _ :: Structure-Level-Declaration →
        action [binding | escaping with a packet | diverging | changing state]
               [using current bindings | current state] .

(1)  establish ⟦ "structure" $B$:Structure-Binding ⟧ =
        establish $B$ .

(2)  establish ⟦ "functor" $B$:Functor-Binding ⟧ |
        establish $B$ .

• establish _ :: Structure-Binding →
        action [binding | escaping with a packet | diverging | changing state]
               [using current bindings | current state] .

(3)  establish ⟦ $I$:Structure-Identifier "=" $S$:Structure-Expression ⟧ =
        structure-evaluate $S$ then
        bind strid $I$ to the given structure .

(4)  establish ⟦ $I$:Structure-Identifier ":" $G$:Signature-Expression
                   "=" $S$:Structure-Expression ⟧ =
        structure-evaluate $S$ then
        bind strid $I$ to (the given structure
             restricted to the interface described by $G$) .

• establish _ :: Functor-Binding →
        action [binding] [using current bindings] .

(5)  establish ⟦ $I_1$:Functor-Identifier
                   "(" $I_2$:Structure-Identifier ":" $G$:Signature-Expression ")"
                   "=" $S$:Structure-Expression ⟧ |
        bind funid $I_1$ to the functor of the closure of the abstraction of
             |furthermore bind strid $I_2'$ to (the given structure
             |      restricted to the interface described by $G$)
             |hence structure-evaluate $S$ .
```

Fig. 13. AS specification of dynamic semantics of SML declarations (extracts).

7 Conclusion

The goals set out in the introduction have largely been achieved:

1. The AS specification of the dynamic semantics of SML is complete. It has demonstrated that AS is very suitable for specifying the dynamic semantics of functional programming languages in ML's category, i.e., polymorphic and strict.[12] Also, AS has no difficulty in coping with ML's module layer, which is more advanced than that found in any other major programming language.
2. The AS specification of the static semantics of ML's core layer is more-or-less complete. This has demonstrated the versatility of the unified-algebraic notation, even to the extent of mimicking a relational specification. No resort to action notation proved necessary. An AS specification of the static semantics of ML's module layer, which would undoubtedly be challenging, has not yet been attempted.
3. A systematic comparison of the AS specification of SML with the NS specification [2] is highly revealing. As expected, the AS specification is much more readable than the NS specification, although less concise. Unexpectedly, the NS specification is partly informal or semi-formal, whereas the AS specification is completely formal. The informal state and exception conventions were essential to keep the NS specification tractable. Without them, the NS specification of the dynamic semantics would have been no more concise, and vastly less readable, than the corresponding AS specification.

The current draft of the AS specification of SML is available for inspection and comment. See:

```
www.dcs.gla.ac.uk/~daw/publications/SMLAS.ps
```

References

1. Brown, D.F., and Watt, D.A.: JAS – a Java action semantics, in these Proceedings (1999).
2. Milner, R., Tofte, M., and Harper, R.: *The Definition of Standard ML*, MIT Press, Cambridge, Massachusetts (1990).
3. Milner, R., and Tofte, M.: *Commentary on Standard ML*, MIT Press, Cambridge, Massachusetts (1991).
4. Mosses, P.D.: *Action Semantics*, Cambridge University Press, Cambridge, England (1992).
5. Mosses, P.D.: A modular SOS for action notation, in these Proceedings (1999).
6. Mosses, P.D., and Watt, D.A.: Pascal action semantics — towards a denotational description of ISO Standard Pascal using abstract semantic algebras, Computer Science Department, Aarhus University, Denmark (1986).
7. Watt, D.A.: An action semantics of Standard ML, in *Mathematical Foundations of Programming Language Semantics* (ed. M. Main *et al.*), LNCS 298, Springer, Berlin (1987), 572–598.

[12] Specifying a lazy functional programming language would be a very different matter.

8. Watt, D.A.: *Programming Language Syntax and Semantics*, Prentice Hall International, Hemel Hempstead, England (1991).
9. Watt, D.A.: Why don't programming language designers use formal methods? in *XXIII Seminário Integrado de Software e Hardware* (ed. R. Barros), Universidade Federal de Pernambuco, Recife, Brazil (1996).

Recent BRICS Notes Series Publications

NS-99-3 Peter D. Mosses and David A. Watt, editors. *Proceedings of the Second International Workshop on Action Semantics, AS '99,* (Amsterdam, The Netherlands, March 21, 1999), May 1999. iv+172 pp.

NS-99-2 Hans Hüttel, Josva Kleist, Uwe Nestmann, and António Ravara, editors. *Proceedings of the Workshop on Semantics of Objects As Processes, SOAP '99,* (Lisbon, Portugal, June 15, 1999), May 1999. iv+64 pp.

NS-99-1 Olivier Danvy, editor. *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, PEPM '99,* (San Antonio, Texas, USA, January 22–23, 1999), January 1999.

NS-98-8 Olivier Danvy and Peter Dybjer, editors. *Proceedings of the 1998 APPSEM Workshop on Normalization by Evaluation, NBE '98 Proceedings,* (Gothenburg, Sweden, May 8–9, 1998), December 1998.

NS-98-7 John Power. *2-Categories*. August 1998. 18 pp.

NS-98-6 Carsten Butz, Ulrich Kohlenbach, Søren Riis, and Glynn Winskel, editors. *Abstracts of the Workshop on Proof Theory and Complexity, PTAC '98,* (Aarhus, Denmark, August 3–7, 1998), July 1998. vi+16 pp.

NS-98-5 Hans Hüttel and Uwe Nestmann, editors. *Proceedings of the Workshop on Semantics of Objects as Processes, SOAP '98,* (Aalborg, Denmark, July 18, 1998), June 1998. 50 pp.

NS-98-4 Tiziana Margaria and Bernhard Steffen, editors. *Proceedings of the International Workshop on Software Tools for Technology Transfer, STTT '98,* (Aalborg, Denmark, July 12–13, 1998), June 1998. 86 pp.

NS-98-3 Nils Klarlund and Anders Møller. MONA *Version 1.2 — User Manual*. June 1998. 60 pp.

NS-98-2 Peter D. Mosses and Uffe H. Engberg, editors. *Proceedings of the Workshop on Applicability of Formal Methods, AFM '98,* (Aarhus, Denmark, June 2, 1998), June 1998. 94 pp.